Coding an Architecture Style

A practical guide to learn **Software Architecture** by coding in Java

Learn fundamental concepts of Software Architecture by coding a **Layered**, **Hexagonal**, **Modular** and **Microservice** Architecture Style

+ A pragmatic study of Monolith and Distributed Structures, Architectural Views and the Availability and Scalability Quality Attributes

Enrique Pablo Molinari

Coding an Architecture Style

A practical guide to learn Software Architecture by coding in Java

Enrique Pablo Molinari

This book is for sale at http://leanpub.com/codinganarchitecturestyle

This version was published on 2020-12-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 Enrique Pablo Molinari

Contents

Feedback . . . . . **i**

I Introduction 1

1. Introduction **2**
Another Book about Software Architecture? 3
What Will I Learn? 3
Who Is This Book For? 5

2. Software Architecture: Fundamentals **6**
What is Software Architecture? 6
Visualizing Software Architecture 7
4+1 View Model 10
Structures and Views (SEI) 11
UML Diagrams 12
C4 Model 12
Conclusion 15
Software Architecture Styles 16
Design vs Architecture 16

II Fundamentals and Implementation 19

3. Modules, Layers and their Architecture Styles **20**
Namespaces, Packages, Layers and Modules 20
Logical vs Physical Separation 20
Java Module System 25
Horizontal vs Vertical Partitions 25
Why Do We Split an Application? 25
Modules vs Layers 26
Layered Architecture 26
Definitions and Style 26
Implementation 27
Hexagonal Architecture 51

Definitions and Style . . . 51
Implementation . . . 52
Inverted Layered Architecture . . . 58
Style and Implementation . . . 58
Modular Architecture . . . 69
Definitions and Style . . . 69
Implementation . . . 72

4. Monolithic and Distributed Architecture . . . 84
Monolithic Architectures . . . 84
Definitions . . . 84
Modular Monolith . . . 84
Distributed Architectures . . . 86
Reasons to Distribute . . . 86
Partial Failures . . . 87
Transactions . . . 91
Distributed Monolith . . . 92
Microservices . . . 94
Modular Architecture vs Microservices . . . 113

III Quality Attributes . . . 115

5. Availability . . . 117
Introduction . . . 117
Calculating the Availability . . . 117
Other Issues Affecting Availability . . . 125
Availability in Microservices . . . 126

6. Scalability . . . 127
Scaling Web Applications . . . 127
Scaling Databases . . . 130
CAP and PACELC Theorems . . . 131
Scaling Reads . . . 132
Sharding: Scaling Writes . . . 134

Feedback

I would love to hear any comment about this book. Email me to enrique.molinari@gmail.com.

I Introduction

1. Introduction

In 1986 Frederick Brooks wrote "No Silver Bullet"[1] where he explains that there are no solutions, silver bullets, to deal with the essential difficulties of software development such as complexity. He described software projects as werewolves, because at the beginning they look familiar but eventually they become into something horrible. But as he describes, for werewolves you have silver bullets, but unfortunately solutions like that do not exist for software projects. However, he mentions **great designers** as one of the promising solutions. He said, "the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort". Today, 34 years from that paper, building **simple** and **clean** software structures is still a hard goal to achieve for software developers.

Over the years, different terms became popular, reflecting the frustration generated by software developers with the tedious work of updating and maintaining code as new requirements are added over time. In 1997 the expression "Big Ball of Mud"[2] became popular and even earlier the term "spaghetti-code" was used. Both referring to software systems without a clear definition of their structure and where all the elements are connected to each other, without an order or clear rules. This obviously makes it extremely difficult to modify them.

In 1999 a concept emerged that would define these insights: software entropy[3]. Meaning that software systems, after several years of maintenance and new added requirements, become something very complex to change. As a general rule, when entropy reaches the software system, that is when frustrated programmers begin to evaluate other job offers, since making changes there is so complicated that it involves allocating overtime, outside of working hours, neglecting other aspects of everyday life. On the other hand, in cases where there is a budget the company decides to start in parallel the re-implementation of the same system with "best practices" or modern technology and thus begins a new iteration towards the (almost always) inevitable entropy.

Years go by, new terms emerge, but the problems and complications remain the same. How to structure a software system in such a way that it allows us to modify it without great complications after several years?

Throughout this book, different practices, concepts, and implementations that seek to answer this question will be presented.

I believe that producing a software system that does not become complex to modify over time is a highly sought after goal (almost unattainable) for every software programmer, designer or architect. Much of what is exposed in this book is dedicated especially to trying to prevent software entropy from winning over us.

Another Book about Software Architecture?

In 1994 David Garlan and Mary Shaw[4] wrote a technical report on the importance of software architecture as an emerging discipline. They said:

"As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem."

Since that technical report to today dozens of books on software architecture have been published. However, most of them lack code examples and implementation. It is usual to find in these books the explanation of the layered architecture style with a beautiful diagram, but no mention is made of how to represent the layers in source code, that is, how they are implemented. From my point of view, this is not a minor omission. I think it is necessary to clearly express and exemplify the set of syntactical constructions that can be used to define a layer and its limits. Today, mainstream programming languages that belong to the same paradigm have a very similar set of syntactical constructions that we can use, with which it is possible to explain these concepts using any language and it will be applicable in an analogous way to any other.

What Will I Learn?

When Professor Garlan and Professor Shaw talk about the difficulties of designing and specifying the general structure of a software system, they mention protocols for communication, synchronization, and data access. Assignment of functionality to design elements, physical distribution, composition of design elements, scaling and performance. This is the next level of design: the **software architecture**.

In line with this definition, the following image shows four diagrams of the *same application* with the intention of visualizing that next level of design, the architecture:

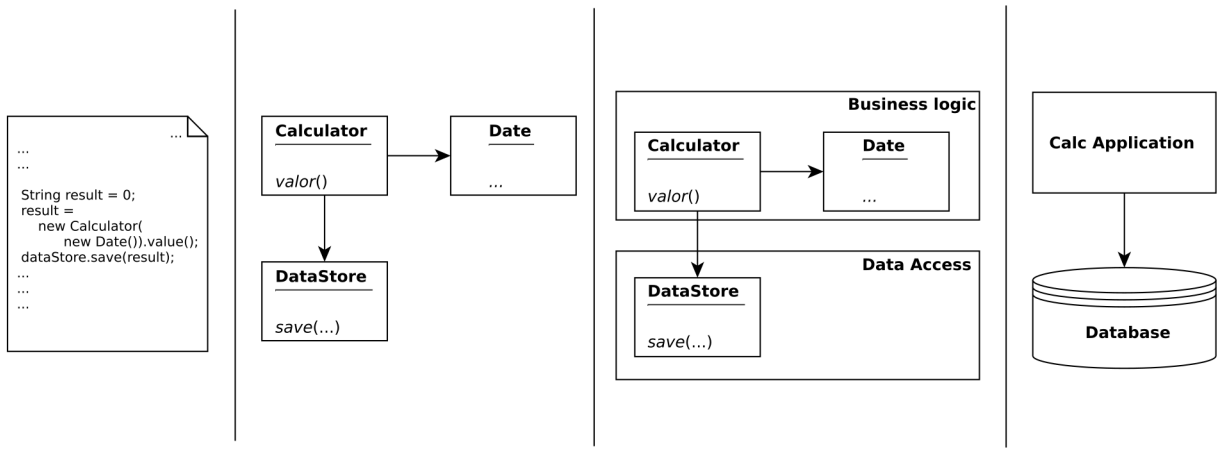

Figure 1.1: Code, Design and Architecture

The first level is clear, software is written using programming languages with specific syntax and semantics, within a specific paradigm. In the next level of abstraction, an object-oriented design is observed, expressed in a class diagram. Here again, no doubt arises, we are talking about design. The third level of abstraction is where some questions begin to appear: is this architecture, or is it design? What exactly does this application diagram describe? Can we have a syntactical construction to map the *Business Logic* and *Data Access* boxes into source code? For now we say that it represents the layered structure of the application, where classes are grouped in each layer according to whether they are related to business logic or data access. Where in addition, the classes of the business logic layer depend on some classes of the data access layer, and not the other way around. In the fourth and last diagram it is possible to observe a higher level of abstraction than the previous one. In this case, *Calc Application* represents the application at *runtime* and *Database* represents the persistent storage at *runtime* too. The arrow indicates an inter-process communication between them.

This book is dedicated to understand, describe, and implement software structures that belong to the last two levels of abstraction illustrated in figure 1.1.

In addition, in this book we will answer these questions:

What is software architecture? Where does design end and architecture begin? How to draw the architecture of a system? How to design a system with clean architecture concepts? What are layers? What about modules? What is physical and logical separation? Am I really implementing a layered architecture? How can I verify if the architecture rules are accomplished? Are monolithic architectures a bad practice? How can I build a modular architecture? How can I build a Microservice architecture? What is high availability? Single point of failure? How do you scale an application to better support growing demand?

This book does not include detailed discussions of all the different styles and patterns of architecture

that exists. It is dedicated to studying and **implementing** in the Java language those styles that I consider most used in enterprise applications. On the other hand, once the exposed concepts are understood, the reader will be able to handle without inconvenience any other style of architecture not included in this book.

Who Is This Book For?

This is a book for software developers. If you also have gone through the bad experience of maintaining a complex application with spaghetti code and which in turn required releases with new features every four weeks, then you will better understand the importance of having a solid architecture.

If you have experience as a developer creating good object-oriented designs, based on tips found in books like Design Patterns[5], Clean Code[6], Refactoring[7], Test Driven Design[8], but you have doubts about how to structure an application, how to recognize different structures, how to diagram the structure of an application or suddenly you find yourself in the position of architect where you also become responsible for availability, performance, scalability, among others questions, then this book is for you.

Notes

1 No Silver Bullet, Essence and Accident in Software Engineering. Frederick P. Brooks. 1986.

2 Big Ball of Mud. Brian Foote and Joseph Yoder, Fourth Conference on Patterns Languages of Programs Monticello, Illinois, September 1997.

3 The Pragmatic Programmer: From Journeyman to Master. Andrew Hunt, David Thomas. 1999.

4 An Introduction to Software Architecture. David Garlan, Mary Shaw. CMU Software Engineering Institute Technical Report. 1994.

5 Design Patterns: Elements of Reusable Object-Oriented Software. 1994.

6 Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin. 2008.

7 Refactoring: Improving the Design of Existing Code. Martin Fowler. 1999.

8 Test Driven Development: By Example. Kent Benck. 2002.

2. Software Architecture: Fundamentals

In this chapter we cover fundamental concepts of software architecture. We begin with those definitions that will allow us to provide theoretical support to later understand the following chapters. We will define software architecture, how it is represented in a diagram for analysis, and how it differs from software design. We will study the different levels of abstraction with which we represent different *views* of the same application, analyzing the proposals of different authors.

What is Software Architecture?

If you have read more than one book or article on software architecture you will know that there is not a single definition of software architecture[9]. However, some important words of each definition are common to the many existing ones. The one I consider most important is **structure**. Let's review a simple and clean definition of software architecture that will guide the rest of this book.

Software Architecture studies and establishes how a software product or system is **structured**. The **structure** comprises **software elements**, their **properties** and the **relationships** between them. We call *software element* to any piece of software that could range from a function, procedure, class, module, layer and package to applications, data repositories, execution environments (web servers, containers, etc.), API gateways, queue systems, etc.

As you can see, there is a long list of software elements that are used to structure a system. For the moment, let's understand that these elements are what we use to express the architecture of a system, that is, to express its structure. Later we will discuss its use in detail.

So, why is it important to understand and visualize the structure of a system? Mainly to be able to understand what architectural characteristics a software system has or what characteristics we want to give to a software system. In the design of any system there are functional requirements that specify the functionalities that the system must provide, but architects (and developers as well) must bear in mind other factors when designing a system. These factors are presented as *quality attributes*[10]. To mention some of these:

Monitorability: Does the system offer options to verify its status? How simple is it to do it?
Scalability: How can the system respond to an increase in the demand?
Availability: If any software element of the system fails, is the end user still getting responses from the system?
Maintainability: How simple is it to migrate from one database vendor to another? When I need to make a change, is it clear where those changes are going to impact?

Quality attributes are also known as *non-functional requirements* or *architecture characteristics*[11]. Throughout part II of this book we will study software structures whose objective is to promote maintainability. In part III we will study other quality attributes like scalability and availability.

Visualizing Software Architecture

We describe the software architecture of a system by drawing its elements and the relationships between them. We draw the software elements, for example, as rectangles and relationships with arrows. What do these rectangles represent? Running applications? classes? modules? And the arrows? compile dependencies? runtime dependencies? Let's look at the next architecture diagram.

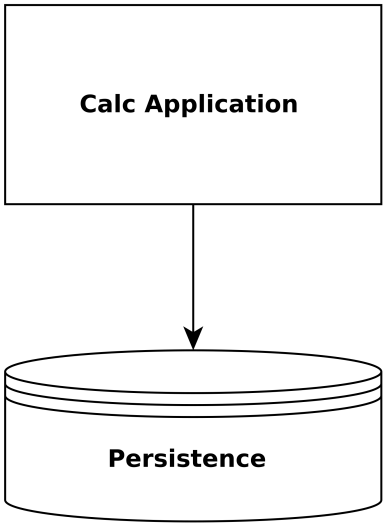

Figura 2.1: Architecture: Runtime Elements

I have said, in the previous chapter, that in this diagram the **Calc Application** element represents the application at **runtime** and the **Persistence** element represents the persistent storage unit at runtime as well. Here we mention a very important point of the diagram. It is a diagram that represents a software system at runtime. But what kind of application is it? Is it web, desktop, mobile? Is the persistent storage a relational database? Let's improve the diagram a bit more.

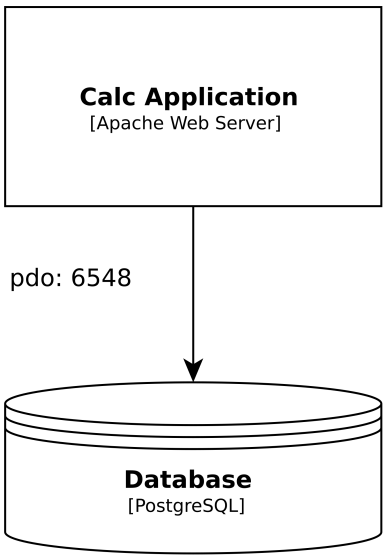

Figura 2.2: Architecture: Runtime Elements

Now, we can see that it is a Web application running on an Apache* Web Server. Use PDO† to communicate with an instance of PostgreSQL‡ through the port 6548. This is a diagram representing the architecture of an application at runtime. With this diagram we could analyze certain quality attributes that we will see in detail later. In principle it shows us that to verify the health of the system we must monitor an Apache service and a PostgreSQL service. An important source of information for the operations team.

Now you might wonder, where is the MVC§ structure represented in that diagram? The answer is that it is not. To represent that, we need another diagram, like the one shown below.

*https://httpd.apache.org/
†https://www.php.net/manual/en/book.pdo.php
‡https://www.postgresql.org/
§https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

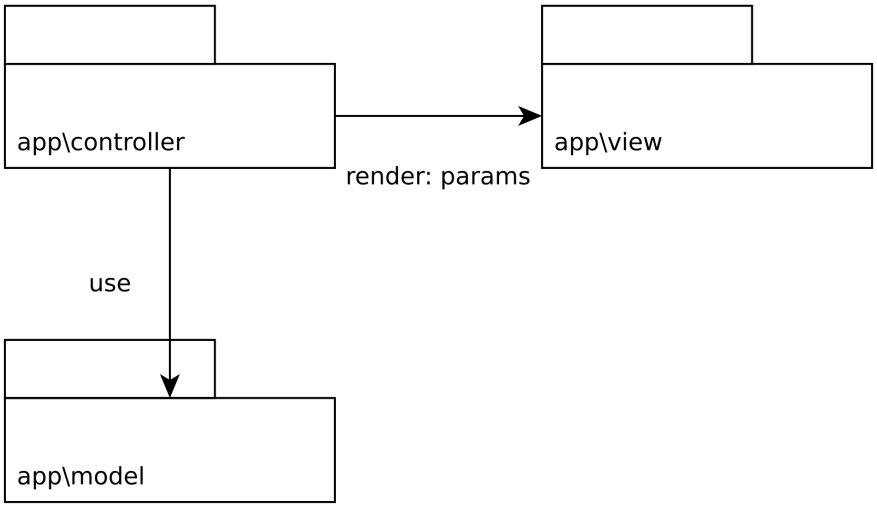

Figure 2.3: MVC Architecture

This is an architecture diagram where the software elements represent **namespaces** (namespaces*) that group classes intended to a common goal. The **namespaces** are equivalent to **packages** in the Java language. The relationships between them represent their dependencies, that is, *app\controller* depends on *app\model* and *app\view*, and the latter have no dependencies. Classes within *app\controller* use classes from the *app\models* to invoke business logic. With the response they obtain talk to *app\view* classes, in charge to transforming input into HTML.

It is important to note that this diagram imposes certain **architecture rules**. For example, from *app\view* it is not allowed to use classes from *app\controller* or *app\model*. The main objective of these rules is to enhance the *modifiability*, to make it clear where the changes we make impact. For example, it allows me to understand that if we wanted to modify some class of the *app\view* package I should check where it impacts in the *app\controller* package but it would not be necessary to verify the *app\model* package.

So far then we have a diagram that represents the structure of the application at runtime (Figure 2.2) and another diagram that represents its package structure along with its dependencies (Figure 2.3). The latter is associated with **architecture decisions** that we assume as a development team.

Is there then more than one diagram that represents the architecture of a system?

Yes, we have different **views**[12] of the architecture of the same system, each view represents a different abstraction of our system that allows us to visualize and reason about different characteristics. In turn, the different views are used to communicate technical decisions to different teams. For example, the package diagram is primarily intended for the development team, while the runtime diagram also includes the operations team, who are in charge of verifying its health while in production.

How many different types of views do we have?

*https://www.php.net/manual/en/language.namespaces.php

It depends, several authors present their proposal. Let us review below the proposals that we consider most outstanding. We will look for similarities between them and then summarize those that we will use throughout this book.

4+1 View Model

One of the first proposals was published in 1995 by Philippe Kruchten, with his model called 4+1[13]. The model presents four different views, plus one called Scenarios that illustrate the system. Each view allows to analyze different issues of the system and also each one is intended for different stakeholders. The following figure illustrates the different views as expressed by Kruchten's work.

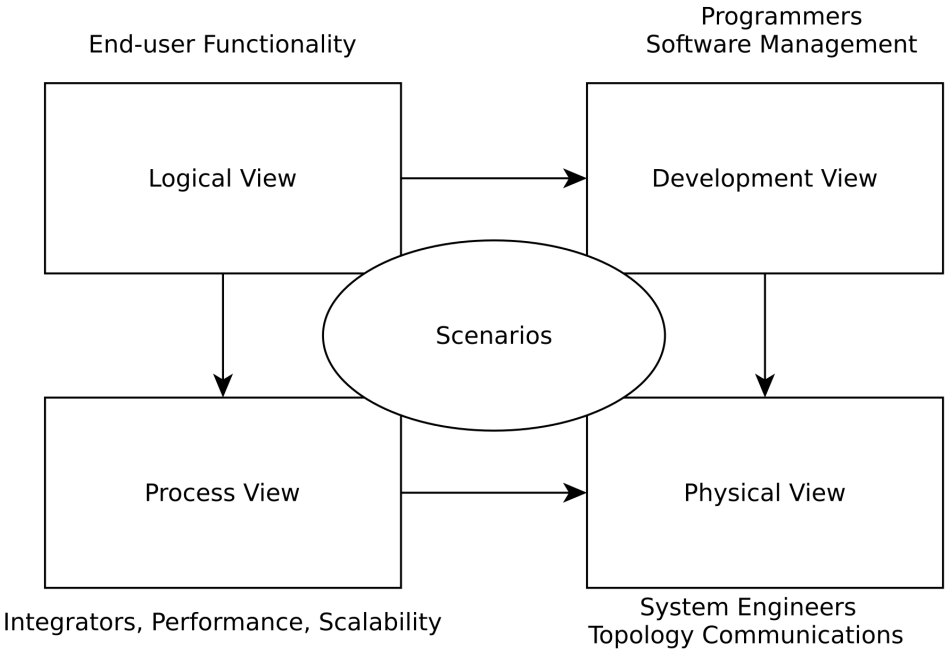

Figure 2.4: 4+1 Model

In the figure 2.4, we can see that there is a **logical view** which is intended for end users. The diagrams in this view focus on presenting the functionalities offered by the system using, for example, class diagrams or entity-relationship diagrams. It is important to note that only those important classes should exist in the diagram or in case of very large systems, it could be necessary to generate abstractions that group classes into categories.

Then we can see the **process view**. By *process* they are referring to the software elements at runtime, which can be stopped or started. This view is equivalent to the one presented in the figure 2.1.

The **development view** raises diagrams of layers, modules and subsystems in a way that allows us to direct functional requirements to specific modules, layers or subsystems. At the same time, if we have different development teams, it helps us in planning the software project, assigning each

team the responsibility for a certain module, layer or subsystem. This view is equivalent to the one presented in the figure 2.3.

The **physical view** refers mainly to showing how the software elements are assigned to hardware elements.

Finally the fifth view, **Scenarios** is dedicated to show how specific use cases are executed by the different software elements.

Structures and Views (SEI)

In one of the Software Architecture "bibles", the book Software Architecture in Practice[14], from the Software Engineering Institute (SEI), the authors define *view*, differentiating it from *structure*, in the following way:

"A view is a representation of a structure. Architects design structures. They document views of those structures".

According to them, architectural structures can be divided into three major categories:

Module Structure: Refers to how the functionality of a system is divided into abstractions (modules, packages, layers, classes, etc). To design structures of this type, we must first understand what syntactical constructions exist in the programming language that we are using to represent this kind of structure. In small systems the classes and their relationships could be the software elements to represent this structure, but in larger systems we need software elements of a higher level of abstraction than a class, function or procedure. For example in PHP the namespaces, in Java the packages and even the modules from Java 9. Using these abstractions we group classes using some criteria (we will talk about this later) and expose interfaces in order to hide implementation details. This type of structure can be documented using the logical and development view of the Kruchten model.

Components and connectors: *components* are the software elements at runtime and *connectors* are the way to connect components, how they communicate. Examples of components could be services, data repositories, applications, anything that can be started and stopped. Examples of connectors are sockets, https communications (REST), shared databases, etc. A component can map one to one to a single module, or it can map to multiple modules from the previous category. To be clear, let's distinguish between the two terms well: the components are software elements at runtime, deployed in a process separated from each other whose communication is in the form of inter-process communication (IPC). While modules are software elements that communicate via their interfaces within running all inside the same process communicating in the form of in-process communication. This type of structure can be represented using the process view of the Kruchten model.

Allocation structures: Refers to decisions of how software elements will relate to hardware elements and also development teams. For example, on which physical or virtual machine each component runs, or which development team is responsible for each component. It is also important to represent the environment we are talking about. For example, in which directory does the system store files in the testing environment and in the production environment. This type of structure can

be documented using Kruchten's physical view to represent components assigned to hardware, and Kruchten's development view to represent components assigned to development teams.

UML Diagrams

UML provides different diagrams to visualize the architecture of a software system. Each diagram offers its notation, not covered in this book, but can be found in Martin Fowler's UML Distilled book[15].

Related to the different structures we were studying, for the module structures we have the diagram of **classes** and the diagram of **packages** of UML. The packages, as mentioned above, correspond to namespaces that we find in languages such as PHP or .NET. The UML also proposes the **component** diagram to represent components and connectors structures. In addition, in initial versions of UML the components were the elements to be deployed on hardware equipment. Later it was modified incorporating the elements that they call *artifacts* (for example WARs, JARs, DLLs, PHARs, etc) which are the ones that are finally deployed (allocated) to hardware. Finally, UML offers the *deployment* diagram that can be used for allocation structures.

C4 Model

Simon Brown created the C4* model to visualize software architectures. Its name is given by the acronyms of the different abstractions of the structures that it allows to diagram: **Context**, **Container**, **Component** and **Code**. Of course, it has many similarities with the other proposals, however, it is presented with examples on traditional information systems unlike the SEI bibliography that is exemplified based on its experience in military systems (motor control, embedded systems and real time)[16], which makes it more complex to understand for those who do not know this domain and its characteristics.

On the other hand, unlike the other proposals, C4 is presented in levels of abstraction. Starting at the highest level, we find the *software system*, represented by the **Context** diagram and where each subsequent level of abstraction corresponds to one step further in detail of the previous level.

In the order of level of abstraction, from the most abstract to the most concrete one, the four types are described:

Context: This diagram represents the *software system* and the relationships with external systems. External systems are those for which we are not responsible for their maintenance. We also include in this diagram the different types of users who use the software system.

Containers: By doing "zoom +" in the box that represents the software system, we find the containers. With containers it refers to applications (web, mobile, single-page, desktop, shell scripts, etc.) and data repositories, schemas and databases. It is essentially a *software element* that executes code or stores data. This structure has similarities with the **components** and **connectors** structure of the SEI proposal.

*https://c4model.com/

Components: By "zooming +" in any of the containers we get to the next level of abstraction, the components. This term designates a set of functionalities encapsulated and exposed through a well-defined interface. Here it is necessary to find in the programming language that we are using the syntactical construction, the abstraction, that we will use as a component. It can be classes, in small systems, or the *packages*, *namespace* or *modules* (Java9+) in the case of more complex systems. On the other hand, there is a big difference in the terminology used that unfortunately confuses. Components in the C4 model is a software element that maps to a syntactical construction or abstraction defined in the source code, while for the SEI proposal and in general in all software architecture bibliography, the components represent software elements at runtime.

Code: Finally, doing "zoom +" on any of the components we would have its implementation modeled, for example, with a class diagram. The author advises not to diagram this level unless it is really necessary.

Lets look at figure 2.5, where we illustrate the first three levels of abstraction of the C4 model.

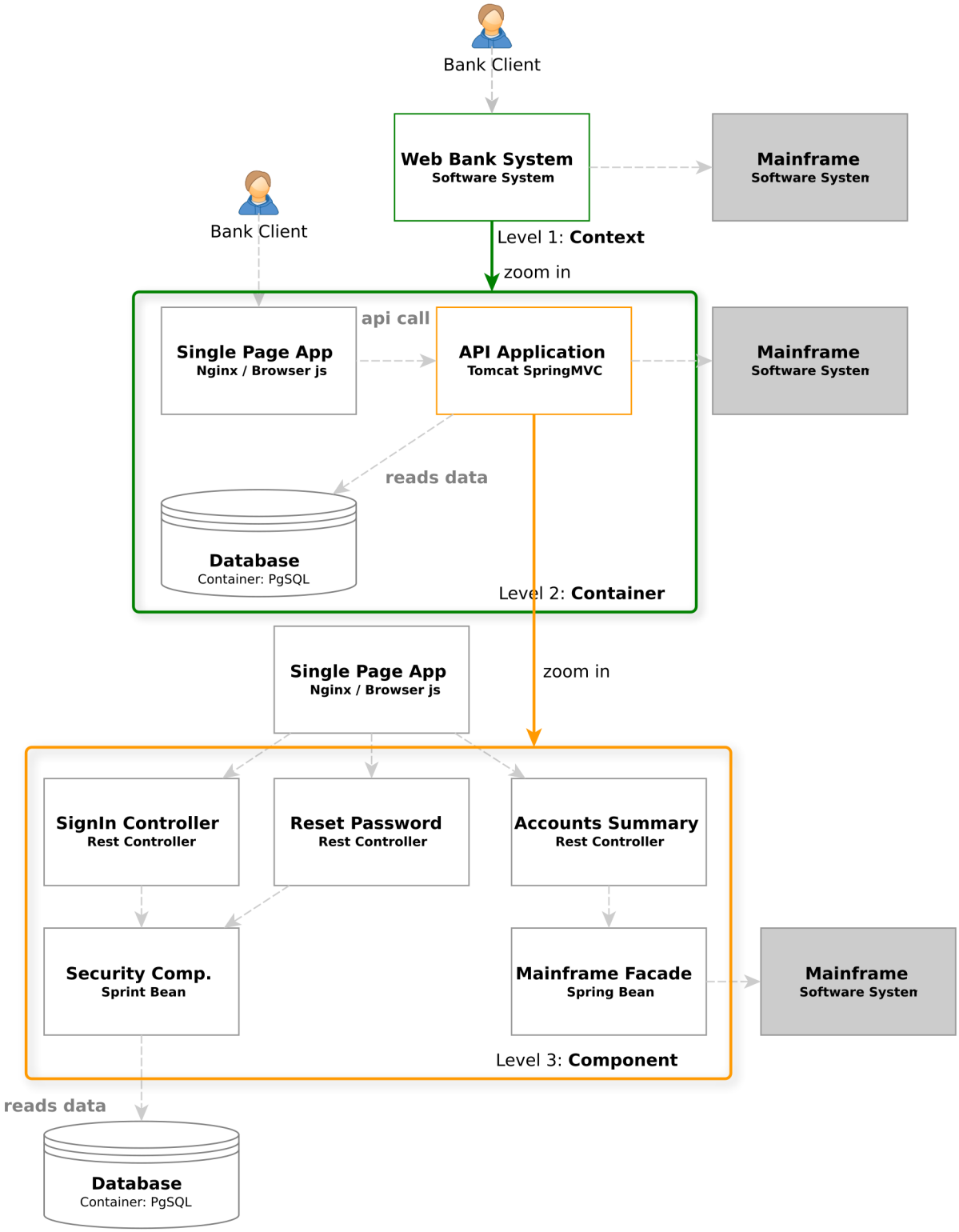

Figure 2.5: C4 Model

In the figure 2.5 we can see three different diagrams belonging to the three firsts levels of abstraction of the C4 model. Three different views of the same system. Starting at level 1, with the **context** diagram. There you can see the "Web Bank System" and its dependencies with external systems, in this case a mainframe. At level 2, going in more detail on the "Web Bank System" box, we have the **container** diagram. There we see a Single-Page application running in a browser, which consumes services from a API Application (SpringMVC) whose execution environment is a Tomcat instance. The latter consumes services from the external system (the mainframe) and accesses a PostgreSQL database. The third and last level of the figure shows the **component** diagram of the *API Application* container. Note that the "zoom +" is performed on that particular container. The box that represents the *Security Comp* component is a package or namespace, whose public interface is consumed by the *SignIn Controller* component. In the book Clean Architecture[17], there is a chapter written by Simon Brown where he explains his proposal about splitting classes by component.

Finally, there are several very interesting tools that allow us to generate the diagrams from the C4 model. We mention some of these tools, Archimate*, C4Builder†, Structurizr‡, PlantUML§.

Conclusion

There are different proposals with different notations to structure and visualize software architectures. However, it is possible to find a common criteria among them, which are what we will use in this book. The objective of this book is that regardless of the terminology used, we can **recognise** the different existing structures to define the architecture of a system and how to represent it using diagrams.

The structures that we will use in this book are:

Modules/Layers: We refer to software elements that map one to one with defined abstractions using syntactical constructions such as classes, packages, namespaces or modules in the programming language that we are using.

It is very important to understand that these diagrams must reflect the reality of the system. This tends to deviate when there is an application built without architecture rules defined, which without exception leads to spaghetti code. But when we show to stakeholders the "architecture" of the system, we draw boxes showing a structure that doesn't really map to our reality in source code.

Containers: Applications or data stores at runtime, as defined by the C4 model (not to be confused with Docker). Typically, containers run within an environment (web server, virtual machine, etc.) in their own process and communication with other containers is in the form of inter-process communication (IPC). While typically the communication between modules or layers is in the form of in-process.

Deployment: Here we identify the physical, virtual and / or container machines (now, Docker), where the software elements (which we have identified in the previous abstraction level) are going

*https://www.archimatetool.com/blog/2020/04/18/c4-model-architecture-viewpoint-and-archi-4-7/
†https://adrianvlupu.github.io/C4-Builder/#/?id=overview
‡https://structurizr.com/
§https://github.com/RicardoNiepel/C4-PlantUML

to be installed. The C4 model has deployment diagrams that are proposed as alternative diagrams, and is based on the UML deployment notation.

To further simplify the different structures that we will use in the book, we can say that we have:

1. Structures that map to abstractions written in source code (**Modules**/**Layers**).
2. Structures that map to software elements at runtime (**Containers**).
3. Structures that map to physical hardware (or virtual machines) (**Deployment**).

In chapters three and four we will focus primarily on understanding and *implementing* structures that belong to modules/layers and container abstractions. In the chapter five and six we will delve into availability and scalability presenting examples with the deployment structure.

Software Architecture Styles

> "Just as good programmers recognized useful data structures in the late 1960s, good software system designers now recognize useful system organizations"[18].

There is an ever-expanding set of styles of software architecture. By *style* we mean ways of organizing or **structuring** a software system. As examples of architecture styles we can mention: Layered, Hexagonal, Service Oriented, Event driven, Microservices, among others. It is not the intention of this book to list and describe every existing style of architecture, but to implement some useful ones.

Design vs Architecture

What do different authors say about the difference between design and architecture?

Grady Booch said: "All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape the system, where significant is measured by the cost of change". Several authors cite this phrase in different books about software architecture[19] [20] [21].

Based on Booch's definition, Simon Brown mention in his book[22] the following items as significant decisions with high cost to change:

The Shape of the system: Web, client - server, monolith, distributed, etc.

The structure of the software system: Modular, layers, etc.

The choice of technology: Programming language, deployment platform, etc.

The choice of frameworks: MVC framework, ORM Framework, etc.

On the other hand, in the book Just Enough Software Architecture[23], George Fairbanks suggests that those who come closest to achieving a clear differentiation between architecture and design are Mary Shawn and David Garlan (whom we have cited earlier in this book) with the following definition:

"As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation. Structural issues include the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. This is the software architecture level of design".

Finally, in the books Clean Architecture[24] and Fundamentals of Software Architecture[25] it is pointed out that architecture corresponds to high-level decisions that are often at a distance from lower-level design decisions. When in fact both should be understood as complementary parts of the same thing: the software we are building. This separation, they say, makes the development team and the architecture team to not work together, and communicate only once at the beginning of the project. So they suggest that it is not possible to speak about design vs architecture, but that both must be built hand in hand.

I cannot agree more with this last statement, the simple idea that architects lock themselves in meetings with the business and neglect their technical skills and their frequent communication with development teams is a terrible decision[26].

All of these explanations sound great, but there is still something missing to really see the difference. I'm particularly interested to visualize that line that differentiates design from architecture, from the technical point of view. In my proposal[27] [28], if we are talking about software elements and their relationships, where these elements map to syntactical constructions in the programming language that we are using, then we are talking about design. And every decision there is a design decision. Whereas any structure composed of software elements at a higher level of abstraction than source code, is architecture. And every decision in that area is an architectural decision. In this way, in general, the structures of modules or layers, and the communication between them in the form of in-process require design decisions. The container and deployment structures and their communication in the form of inter-process require architectural decisions. This proposal establishes a clear line between design and architecture.

Notes

9 What is your definition of software architecture? Carnegie Mellon University.

10 Software Architecture in Practice. 3era Edición. Len Bass, Paul Clements, Rick Kazman. Chapter 4.

11 Fundamentals of Software Architecture: An Engineering Approach 1st Edición. Mark Richards, Neal Ford. 2020.

12 Or called perspective.

13 The “4+1” View Model of Software Architecture. Philippe Kruchten. 1995.

14 Software Architecture in Practice. 3era Edición. Len Bass, Paul Clements, Rick Kazman. 2012.

15 UML Distilled. A Brief Guide to the Standard Object Modeling Language. Third Edition.0 Addison-Wesley Professional. Martin Fowler.

16 Essential Software Architecture, Ian Gorton. Springer-Verlag, 2006.

17 Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Robert C. Martin. 2017.

18 An Introduction to Software Architecture. David Garlan, Mary Shaw. CMU Software Engineering Institute Technical Report. 1994.

19 Software Architecture for Developers. Simon Brown. 2015.

20 Designing Software Architectures. A Practical Approach. Humberto Cervantes, Rick Kazman. 2016.

21 97 Things Every Software Architect Should Know. Richard Monson-Haefel, Kevlin Henney, 2009.

22 Software Architecture for Developers. Simon Brown. 2015.

23 Just Enough Software Architecture: A Risk-Driven Approach. George Fairbanks. 2010.

24 Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Robert C. Martin. 2017.

25 Fundamentals of Software Architecture: An Engineering Approach 1st Edición. Mark Richards, Neal Ford. 2020.

26 copypaste/architect-role

27 copypaste/design-vs-arch

28 stackoverflow/design-vs-arch

II Fundamentals and Implementation

3. Modules, Layers and their Architecture Styles

In this chapter we will talk about the *structures* and *styles* of architecture that belong to the **modules/layers** structures as described in chapter 2.

Namespaces, Packages, Layers and Modules

In chapter 2 we said that the elements that correspond to the **modules/layers** structures should map to syntactical constructions defined in the programming language that we are using. In small applications we use *classes* to structure an application. But as the complexity and the size of the code base grows, we need others **syntactical constructions** to architect an application in order to allows us to see where the changes will impact. It is not the same to make modifications in an application where we have 200 classes in a package (or in several but without any definition of dependencies between them), than to have 200 classes divided into some packages with a solid definition of dependencies between them.

To group classes into modules, programming languages generally provide us with what they call **namespaces** (C#, PHP, etc) or **packages** (Java). Using packages we can group classes according to certain criteria and define architecture rules that govern the relationships between them. The relationships refer to dependencies that certain classes have with other classes located in other packages, which in turn causes the dependency between packages to exist.

Logical vs Physical Separation

When we define modules or layers and their relationships using namespaces or packages, we say that there is a **logical separation** between them. That is, its limits are defined by syntactical elements, but physically all the classes regardless of which namespace they belong, are located in the same physical place. This means that we are able to reference from any class in any package to any other class in any other package. It also means that classes from a third-party library that is only used by a specific package (specific module or layer) might also be accessible to classes from any other package (module or layer).

On the other hand, when we define modules or layers and their relationships using applications that communicate with each other in the form of inter-process (for example through https messages) we are imposing a **physical separation** between them. Physical separation **ensures** compliance with basic architecture rules. To mention a few, modules or layers with physical separation will only be able to communicate between them through their public interface. In addition, the dependencies

(third-party libraries) of a certain module or layer will only be available to the module or layer that requires it.

The following figure illustrates what might happen using logical separation:

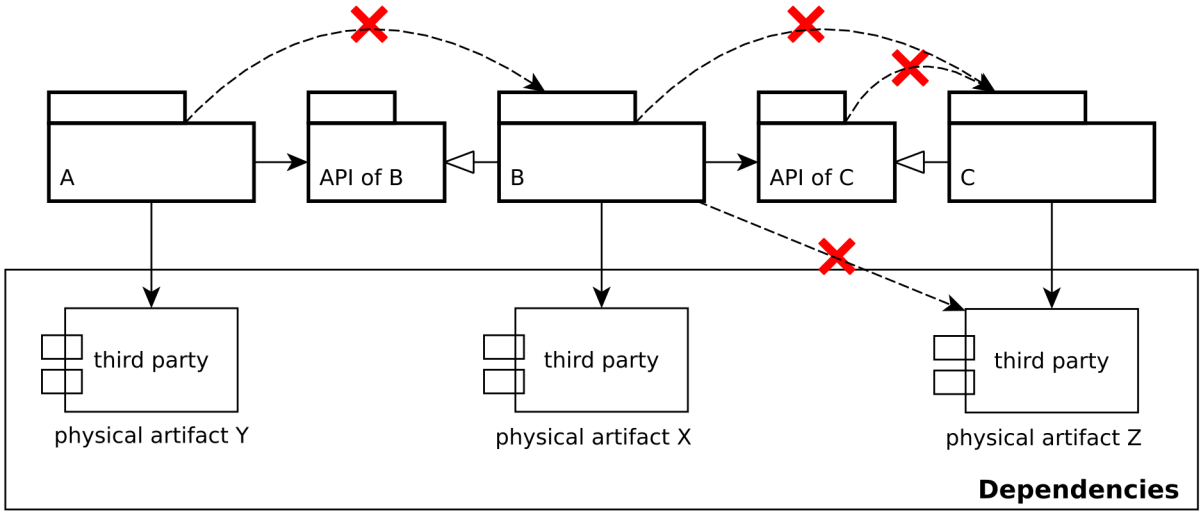

Figure 3.1: Logical Separation

In the figure 3.1 we have three modules/layers, A, B and C, defined using packages. In solid arrows we show the valid dependencies between them according to the architecture rules that we have defined. However, nothing prevents that by mistake or ignorance to have invalid dependencies such as those shown with dotted arrows.

Programming languages have historically provided poor syntactical elements to help us enforce dependencies between namespaces or packages. Which is why when we use logical separation we have to use other tools to help us enforce architecture rules.

Let's see then what options we have to be able to enforce certain architecture rules and still have an application where their modules communicate using just method calls (in-process communication).

The first option would be to split the application with logical separation into five different applications. We will transform each package into an application by taking the source code of a package and their dependencies and put it into a source code project folder. This means that we will change our logical design for a **physical design**. Let's do this with the packages of the figure 3.1. The following figure illustrate that:

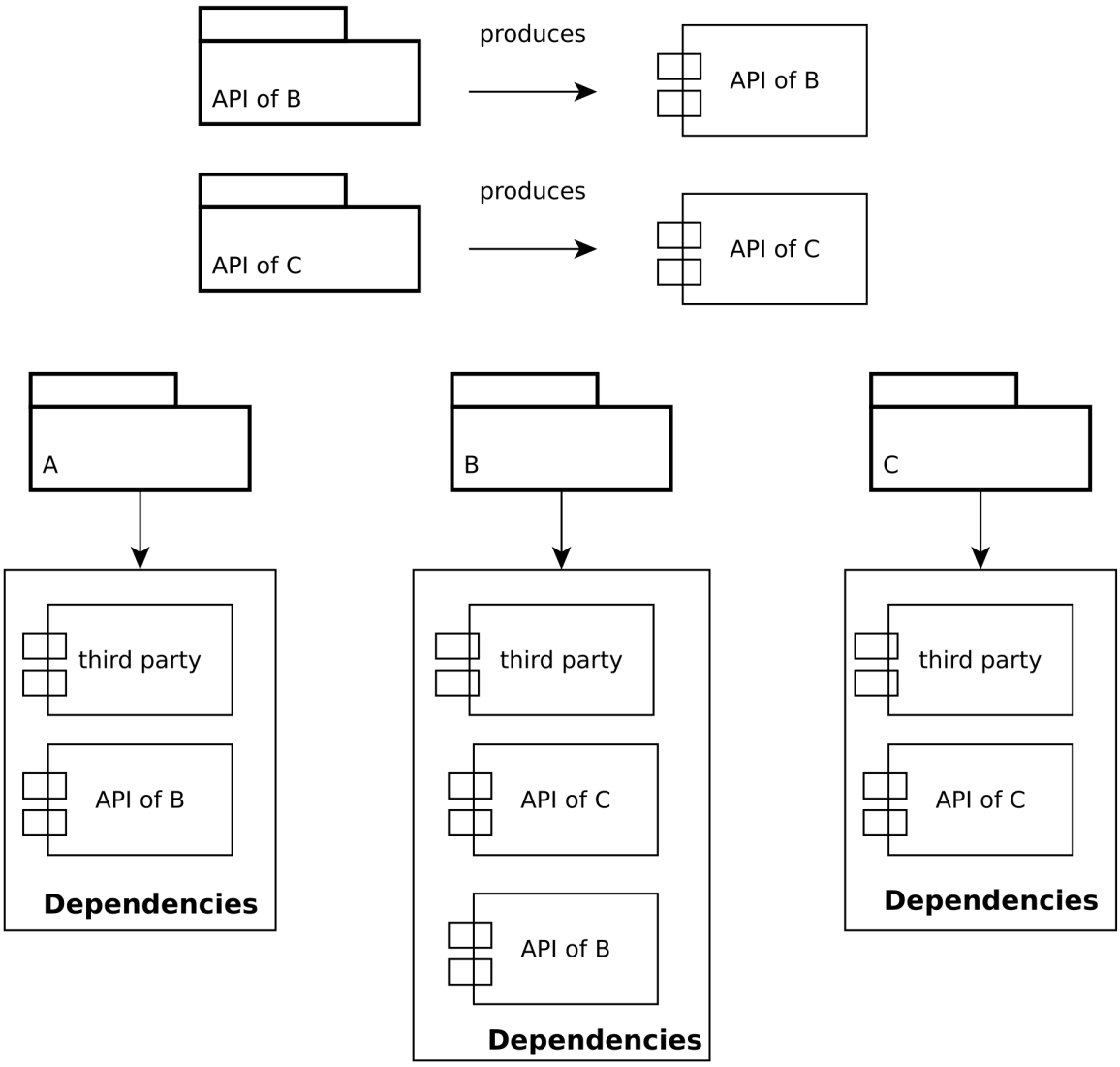

Figure 3.2: Physical Separation

Figure 3.2 shows each package now converted in an application which means that each package now has its own source code project folder with their dependencies. The packages *API of B* and *API of C* (typically containing only interfaces), as shown in the figure do not depend on any other package or third party libraries. Each of these projects generates an artifact (*jars* in Java, *assemblies* in .NET, *gems* in Ruby, etc), which will be part of the dependencies of the other modules (applications) that require it.

In the same figure we can see three more applications, one for each remaining package: *A*, *B* and *C*. Each one with its third-party dependencies and where they also include the artifact generated by the *API of B* and *API of C* as part of their dependencies. Finally, all the artifacts are deployed together, within the same environment to start the application.

Using this way of building the application allow us to have only valid dependencies as defined by figure 3.1 with solid arrows. Note that there is no possibility, for example, for module A to reference a class that belongs to module B. However, as a drawback we are complicating the development environment, the build script and deployment.

The other option, if we want to maintain the logical design - a single source code project folder structured using namespaces or packages - we should add a way to **verify** that all the references between packages are valid. And this verification must execute on every commit.

How can we do this?

One possible option but only available in compiled languages (for example Java or .NET) is to compile each package as isolated as needed (only with their valid dependencies), as if they were really different projects. If they compile then it means that our dependencies are valid.

Assuming the logical design shown in figure 3.1, let's see below how we could validate it using an Apache Ant* script.

Ant Script, Compiling Packages Isolated

```
<target name="compile-api-of-c">
 <mkdir dir="build/classes-api-of-c" />
 <javac sourcepath="" srcdir="src" destdir="build/classes-api-of-c"
   includes="api of c/*.java">
 </javac>
</target>

<target name="jar-api-of-c" depends="compile-api-of-c">
 <mkdir dir="build/jar-api-of-c"/>
 <jar destfile="build/jar-api-of-c/api-of-c.jar"
   basedir="build/classes-api-of-c">
 </jar>
</target>

<target name="compile-c" depends="jar-api-of-c">
 <path id="lib.path.ref">
  <fileset dir="jars-third-party" includes="physical-artifact-z.jar"/>
  <fileset dir="build/jar-api-of-c" includes="*.jar"/>
 </path>
 <mkdir dir="build/classes-c" />
 <javac sourcepath="" classpathref="lib.path.ref"
   srcdir="src/main/java" destdir="build/classes-c"
  includes="c/*.java" excludes="api of c/*.java">
 </javac>
</target>
```

*https://ant.apache.org/

The Ant script has three *tasks* defined by the *target* XML tag. In the first one called *compile-api-of-c*, we compile only the classes belonging to the package "api of c", specifying this in the *includes* attribute of the *javac* XML tag (note that it is necessary to specify the attribute *sourcepath=""* to let *javac* know that only compile classes inside the ones specified by the *includes* attribute). The next task named *jar-api-of-c* generates the *jar* artifact including in it the compiled classes (*.class* files) done by the previous task.

Finally, the *compile-c* task compiles only package *C* (note again the *includes* attribute of the *javac* task on line 23) including the third-party library *physical-artifact-z.jar* plus the classes from the package "API of C" archived in the *api-of-c.jar* artifact. The latter is defined by the *path* XML element on line 16 and is included in the *javac* task with the *classpathref* XML attribute. By applying the same ideas to the rest of the packages we can achieve the verification we need.

Another way to verify package dependencies is using a tool called ArchUnit*. Lets see the next two test cases:

Verifying Dependencies with ArchUnit

```
1  @Test
2  public void packageApiOfCOnlyDependOnItOrJdk() {
3   JavaClasses jc = new ClassFileImporter().importPackages("apiofc", "c");
4    classes().that()
5    .resideInAnyPackage("apiofc")
6    .should()
7    .onlyDependOnClassesThat()
8    .resideInAnyPackage("apiofc","java..")
9    .check(jc);
10 }
11
12 @Test
13 public void packageCOnlyDependOnApiOfCOrJdk() {
14  JavaClasses jc = new ClassFileImporter().importPackages("apiofc", "c");
15
16  classes().that()
17   .resideInAnyPackage("c")
18   .should()
19   .onlyDependOnClassesThat()
20   .resideInAnyPackage("c", "apiofc", "java..")
21   .check(jc);
22 }
```

ArchUnit has a very simple and fluent API as you can see. The first test case verifies that the classes in the *API Of C* package depend only on the JDK, while the second test case verifies that the classes in the *C* package depend only on the *API of C* classes and on the JDK.

*https://www.archunit.org/

In this book we will mostly use ArchUnit[29] to verify dependencies and architecture rules.

Enforcing architecture rules is the only way to avoid spaghetti code. In addition, if in the future is needed, we will be able to move from a logical design to a physical one without much effort.

Java Module System

The Java language in its version 9 has released a module system. By providing the **module** syntactical construction you can create modules which are basically a group of packages. Each module must have a *descriptor* called **module-info**.*java* used to define which packages are *public* and which are *private* to the module. Which packages are *public* but only accessible to certain other modules. Furthermore, each module must indicate in its descriptor which *public* packages it requires from other modules. Another important improvement it offers is that each module has its own *classpath*, called *modulepath*. This means that all the dependencies of a module exist only within its *modulepath* without interfering with the dependencies of other modules, contrary to what happens with packages and the *classpath*, where all the packages of the same application share the same dependencies (with the impossibility of using different versions of the same dependency for different packages).

The rules that we define in the descriptors of the modules are verified at compile time and at run time. If we structure the application of figure 3.1 using the *module* keyword instead of packages, we would have guaranteed compliance with the architecture rules without the need to use other tools. In addition, the Java module system also checks for invalid dependencies created using reflection*, while none of the mechanisms shown before do.

It is not the intention of this book to explain in detail the benefits of Java9 and its proposed module system, if you want more detail you can read the excellent book by Sander Mak[30]. However, we will use it in depth towards the end of the chapter to implement a modular architecture.

Horizontal vs Vertical Partitions

Now we know that we have namespaces, packages, modules (Java9) or even applications to structure an application in **modules** or **layers**. Let's continue with studying the differences between modules and layers and then we will implement their different styles.

Why Do We Split an Application?

When we talk about splitting an application into modules or layers, is with the goal of giving structure to an application. An structure in such a way that we can improve the **modifiability** quality attribute, which means to have clarity of where the changes we make impact.

*https://www.oracle.com/technical-resources/articles/java/javareflection.html

Modules vs Layers

We use the same abstractions, the same syntactical constructions to define modules or layers within an application. These are the ones we mentioned in the previous section, namespaces, packages, modules (Java9 and other languages) and also applications.

One of the most important aspects to define first is whether we are going to make a **horizontal** or **vertical** partition. In general, when we talk about **layers**, we mean a horizontal partition of an application. That is, to divide an application according to **technical** aspects, which are usually related to the user interface, business logic and data access. On the other hand, splitting an application vertically implies dividing it according to **functional** aspects, and for this we usually speak of **modules*** and not layers. Certain classes that solve certain functionality grouped in one module and certain other classes that implement and solve other functionality grouped in another module. If someone outside the development team looked at the module structure of a vertically partitioned application, they could identify the main functionalities of the system (and also your architecture will "scream" their purpose as described by Uncle Bob†). Whereas if it does it in an application with horizontal partition it could not do it. Perhaps they might notice that certain technical aspects are resolved by the development teams that best handle such technology (front-end vs back-end developers as an example).

Let's see then the different structures and styles that allow us to split an application horizontally and vertically.

Layered Architecture

Definitions and Style

The layered architecture style is probably the most widely used style, simplest to understand and apply at the same time. It is one of the oldest styles too. Two very good books, Patterns of Enterprise Application Architecture[31] and Fundamentals of Software Architecture[32], almost 20 years apart from each other, describe the layered architecture style in a similar way.

This style suggests structuring the software elements of an application as horizontal layers, where each layer is dedicated to a certain technical aspect, such as the user interface, business logic and persistence. Within the user interface layer we can find elements for presenting information to users (Windows, HTML, etc.), handling the interaction between the end user and devices such as the mouse, keyboards or touch screens. We also include in this layer software elements that allow us to expose HTTP services (Ex: REST). Whereas the communication between the different layers of this style of architecture is usually in the form of in-process, we could deploy the layers in different machines[33] and make their communication in the form of inter-process. The business logic layer is where we must incorporate validations, calculations and all those algorithms that allow us to verify

*Also called *vertical*.

†Thttps://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

the definitions and rules present in the problem domain that we are modeling. It also decides which services from the persistence layer to invoke. Finally, the persistence layer contains source code that deals with the interfaces and drivers from the storage services.

The layered architecture style can be **closed**, that is, each layer can only access the layer immediately below, or it can be **open** where for example the user interface layer is allowed to access the persistence layer.

Each architecture style imposes certain rules on us. We generally get relaxed about checking these rules and end up with an application with spaghetti code. For this reason it is important to have a verification mechanism for the defined architecture rules, such as those presented in the previous section.

In the next section we are going to go step by step building an application in such a way that it allows us to understand how this style of architecture is implemented.

Implementation

To show how the layered architecture is implemented we are going to use an existing small application written without any architecture rule. During this learning experience we are going to refactor it towards a layered structure.

This application that we will call "RadioA" is used as part of the operation of a radio show. The radio show launches competitions with a certain frequency. The competitions are promoted during the radio show where conditions and the rules are indicated. Listeners must send an email to request the registration to the competitions and an employee of the radio uses the "RadioA" application to register them.

To persist the inscriptions to competitions we have defined the following relational schema:

RadioA's Relational Data Model

```
CREATE TABLE competitor (
   id INTEGER NOT NULL,
   first_name VARCHAR(255),
   last_name VARCHAR(255),
   person_id VARCHAR(15),
   email varchar(50),
   phone varchar(35),
   PRIMARY KEY (id)
);

CREATE TABLE competition (
   id INTEGER NOT NULL,
   description VARCHAR(255),
   rules VARCHAR(1500),
```

```
15     start_date timestamp,
16     inscription_start_date timestamp,
17     inscription_end_date timestamp,
18     PRIMARY KEY (id)
19 );
20
21 CREATE TABLE inscription (
22     id_competition INTEGER references competition(id),
23     id_competitor INTEGER references competitor(id),
24     inscription_date timestamp
25 );
```

The application is developed in the Java language using Swing* to create the user interface. Figure 3.3 shows the registration screen used by the radio employee.

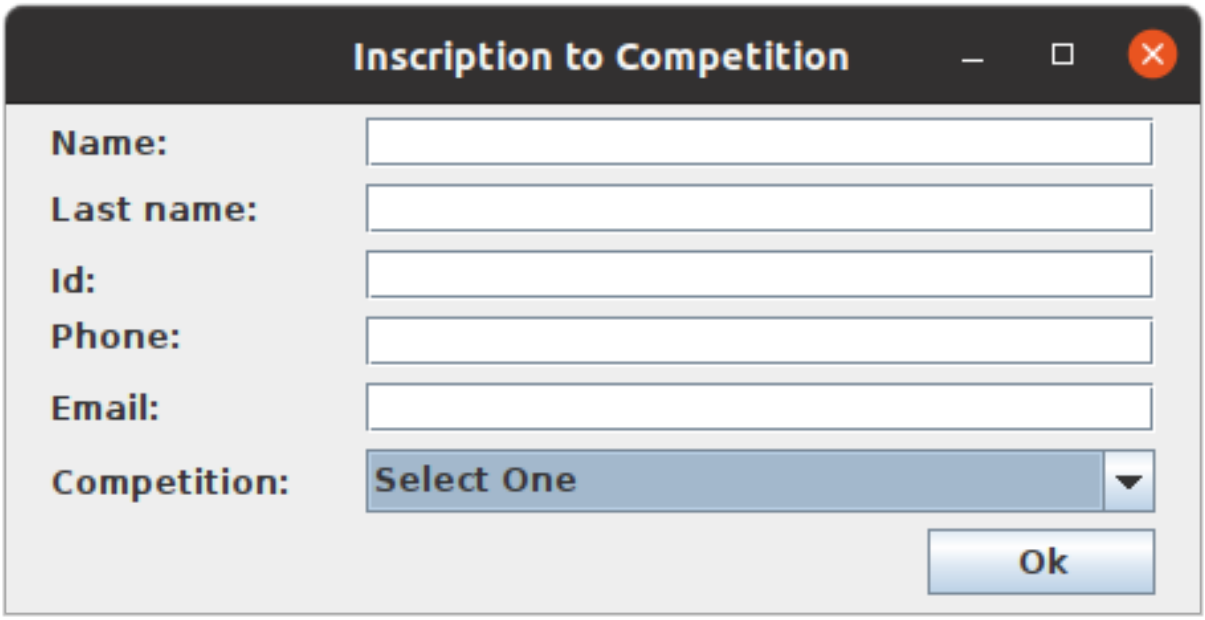

Figure 3.3: RadioA Inscription Window

As can be seen in figure 3.3, the screen has five text fields for the participant's personal data and then a simple listbox of available competitions. By pressing the "Ok" button, the validations are executed and the participant is enrolled (inserting records in the *competitor* and *inscription* tables).

In the figure 3.4 you can see the *containers* diagram (on the left) and the *modules/layers* diagram (on the right) of "RadioA". As the module/layer diagram illustrates, the application is structured in a single package where a single class called *RadioCompetition* implements all the requirements.

*https://en.wikipedia.org/wiki/Swing_(Java)

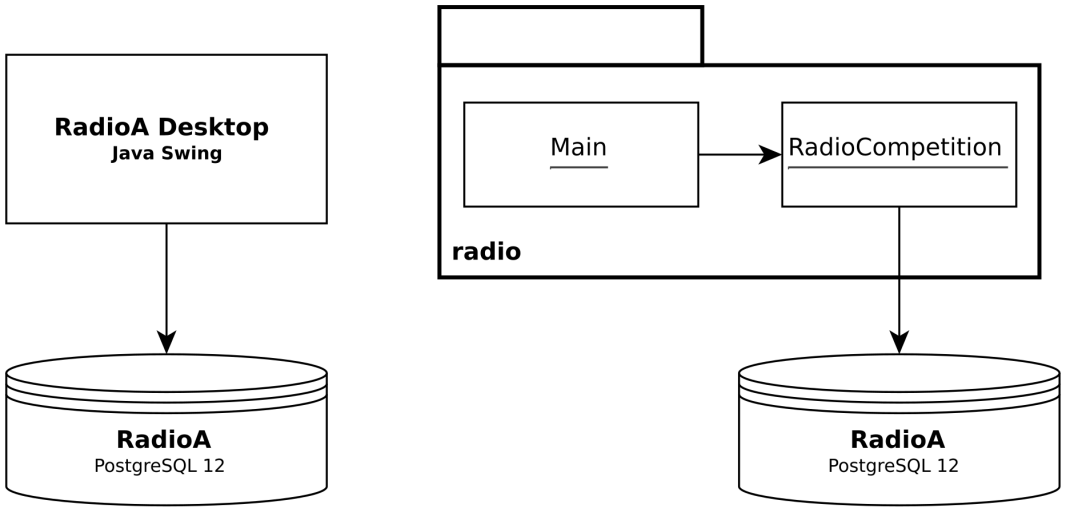

Figure 3.4: RadioA Architecture

Let's look at the sources for the *RadioCompetition* class and then for the *Main* class.

RadioA Application

```
1  class RadioCompetition {
2   private JPanel contentPane;
3   private JLabel lblName;
4   private JTextField txtName;
5   private JLabel lblLastName;
6   private JTextField txtLastName;
7   private JLabel lblId;
8   private JTextField txtId;
9   private JLabel lblPhone;
10  private JTextField txtPhone;
11  private JLabel lblEmail;
12  private JTextField txtEmail;
13  private JComboBox<Item> comboBox;
14  private JButton btnOk;
15  private JLabel lblCompetition;
16
17  public RadioCompetition() throws SQLException {
18   var frame = new JFrame("Inscription to Competition");
19   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20   frame.setBounds(100, 100, 451, 229);
21   contentPane = new JPanel();
22   contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
23   frame.setContentPane(contentPane);
24
```

```
25    formElements();
26    layout();
27    frame.setVisible(true);
28   }
29
30   private Connection connection() throws SQLException {
31    String url = "jdbc:derby://localhost:1527/radiocompetition";
32    String user = "app";
33    String password = "app";
34    return DriverManager.getConnection(url, user, password);
35   }
36
37   private void formElements() throws SQLException {
38    lblName = new JLabel("Name:");
39    txtName = new JTextField();
40    txtName.setColumns(10);
41    lblLastName = new JLabel("Last name:");
42    txtLastName = new JTextField();
43    txtLastName.setColumns(10);
44    lblId = new JLabel("Id:");
45    txtId = new JTextField();
46    txtId.setColumns(10);
47    lblPhone = new JLabel("Phone:");
48    txtPhone = new JTextField();
49    txtPhone.setColumns(10);
50    lblEmail = new JLabel("Email:");
51    txtEmail = new JTextField();
52    txtEmail.setColumns(10);
53    btnOk = new JButton("Ok");
54
55    btnOk.addActionListener(new ActionListener() {
56     public void actionPerformed(ActionEvent e) {
57      var worker = new SwingWorker<Void, Void>() {
58       @Override
59       protected void done() {
60        try {
61         btnOk.setEnabled(true);
62         get();
63         JOptionPane.showMessageDialog(contentPane, "Inscription done successfully !");
64        } catch (InterruptedException | ExecutionException e) {
65         JOptionPane.showMessageDialog(contentPane, "Something went wrong...");
66         throw new RuntimeException(e);
67        }
```

```
68        }
69
70        @Override
71        protected Void doInBackground() throws Exception {
72         btnOk.setEnabled(false);
73         saveInscription();
74         // to make the compiler happy...
75         return null;
76        }
77       };
78
79       if (validations())
80        worker.execute();
81       }
82     });
83
84     lblCompetition = new JLabel("Competition:");
85     allCompetitions();
86    }
87
88    private boolean validations() {
89     if ("".equals(txtName.getText())) {
90      JOptionPane.showMessageDialog(this.contentPane, "Name cannot be empty");
91      return false;
92     }
93
94     if ("".equals(txtLastName.getText())) {
95      JOptionPane.showMessageDialog(this.contentPane, "Last name cannot be empty");
96      return false;
97     }
98
99     if ("".equals(txtId.getText())) {
100     JOptionPane.showMessageDialog(this.contentPane, "Id cannot be empty");
101     return false;
102    }
103
104    if (!checkEmail(txtEmail.getText())) {
105     JOptionPane.showMessageDialog(this.contentPane, "email cannot be empty");
106     return false;
107    }
108
109    if (!checkPhone(txtPhone.getText())) {
110     JOptionPane.showMessageDialog(this.contentPane,
```

```
                            "phone number must be: NNNN-NNNNNN");
    return false;
   }
   if(this.comboBox.getSelectedIndex() == 0) {
    JOptionPane.showMessageDialog(this.contentPane, "Choose a competition");
    return false;
   }
   return true;
  }

  private void allCompetitions() throws SQLException {
   Connection c = connection();
   try {
    PreparedStatement st = c.prepareStatement(
                            "select id, description from competition "
                            + "where inscription_start_date <= ? "
                            + "and inscription_end_date >= ?");
    st.setTimestamp(1, Timestamp.valueOf(LocalDateTime.now()));
    st.setTimestamp(2, Timestamp.valueOf(LocalDateTime.now()));

    ResultSet resultSet = st.executeQuery();

    var competitions = new ArrayList<Item>();
    competitions.add(new Item(0, "Select One"));
    while (resultSet.next()) {
     competitions.add(new Item(
            resultSet.getInt("id"), resultSet.getString("description")));
    }

    this.comboBox = new JComboBox<Item>(
     new CompetitionComboBoxModel(
                    competitions.toArray(new Item[competitions.size()])));
    } finally {
     c.close();
    }
  }

  private void saveInscription() throws SQLException {
   Connection c = connection();
   try {
    c.setAutoCommit(false);
    PreparedStatement st = c.prepareStatement(
                  "insert into competitor(first_name, last_name,"
```

```
154                 +"person_id, email, phone) "
155                 + "values(?,?,?,?,?)", Statement.RETURN_GENERATED_KEYS);
156    st.setString(1, txtName.getText());
157    st.setString(2, txtLastName.getText());
158    st.setString(3, txtId.getText());
159    st.setString(4, txtEmail.getText());
160    st.setString(5, txtPhone.getText());
161    st.executeUpdate();
162
163    ResultSet generatedKeys = st.getGeneratedKeys();
164    generatedKeys.next();
165
166    PreparedStatement st2 = c.prepareStatement(
167                  "insert into inscription(id_competition, id_competitor, "
168                  +"inscription_date) values(?,?,?)");
169
170    st2.setInt(1, ((Item) this.comboBox.getSelectedItem()).id());
171    st2.setInt(2, generatedKeys.getInt(1));
172    st2.setTimestamp(3, Timestamp.valueOf(LocalDateTime.now()));
173    st2.executeUpdate();
174
175    c.commit();
176   } catch (Exception e) {
177   try {
178    c.rollback();
179    throw new RuntimeException(e);
180   } catch (SQLException s) {
181    throw new RuntimeException(s);
182   }
183   } finally {
184    try {
185     c.setAutoCommit(true);
186    } catch (SQLException s) {
187     throw new RuntimeException(s);
188    }
189   }
190  }
191
192  private boolean checkEmail(String email) {
193   String regex = "^[\\w-_\\.+]*[\\w-_\\.]\\@([\\w]+\\.)+[\\w]+[\\w]$";
194   return email.matches(regex);
195  }
196
```

```
197   private boolean checkPhone(String telefono) {
198    String regex = "\\d{4}-\\d{6}";
199    return telefono.matches(regex);
200   }
201  }
```

Main - RadioA Application

```
1  class Main {
2   public static void main(String[] args) {
3    SwingUtilities.invokeLater(new Runnable() {
4     @Override
5     public void run() {
6      try {
7       new Main().start();
8      } catch (Exception e) {
9       // log exception...
10      System.out.println(e);
11     }
12    }
13   });
14  }
15
16  private void start() throws SQLException {
17   new RadioCompetition();
18  }
19 }
```

As you can see, the *RadioCompetition* class has all the application code (github/radioa-no-structure*), mixing concerns of UI, business logic and data access in the same class. If this app were always this size, this wouldn't be so bad. But with some success on the radio show, the app will grow.

As a learning exercise we are going to structure this application in layers. The layers and their relationships that we will implement are illustrated in figure 3.5.

*https://github.com/enriquemolinari/radioa-no-structure

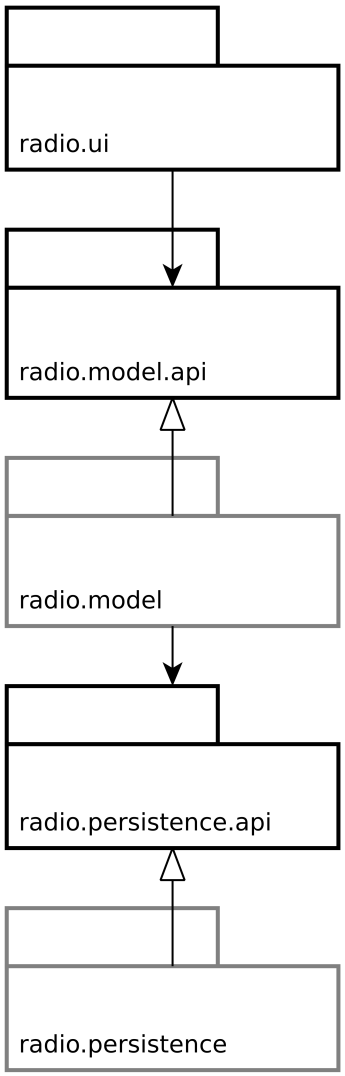

Figure 3.5: RadioA Layered Architecture

The *ui*, *business*, and *persistence* layer are represented by **radio.ui**, **radio.model** and **radio.persistence** packages respectively. As the business and persistence layers expose services, we add two sub-packages of them called **radio.model.api** and **radio.persistence.api** that will contain their public interfaces. So, as the figure 3.5 shows, the *ui* layer depends on the public interfaces located in the *radio.model.api* package, but not on their implementation (*radio.model*). And the same occurs with the dependency from the *business* layer to the *persistence* layer.

So let's start to map this structure into the source code. First we generate the five packages mentioned and also a **main** package where we move the current *Main* class.

Let's look at the "RadioA" sources for the UI class *RadioCompetition*, especially the *allCompetitions*() and *saveInscription*() methods. There we can see some UI related code that are mixed with responsibilities that the business layer should solve. To extract this code, we create the following three

interfaces in the **radio.model.api** package.

Business Logic Public Interfaces

```
1  package radio.model.api;
2
3  public interface RadioProgram {
4   Iterable<RadioCompetition> availableCompetitions();
5   void addInscription(int idCompetition, Competitor c);
6  }
7
8  public interface RadioCompetition {
9   int id();
10  String description();
11 }
12
13 public interface Competitor {
14  String name();
15  String lastName();
16  String phone();
17  String id();
18  String email();
19 }
```

We rename the class radio.ui.*RadioCompetition* to radio.ui.*InscriptionView* and add to the constructor a parameter that allows us to inject an implementation of **radio.model.api**.*RadioProgram* interface.

Constructor Injection in the UI Layer

```
1  package radio.ui;
2
3  public class InscriptionView {
4
5   private RadioProgram radioProgram;
6   ...
7   public InscriptionView(RadioProgram radioProgram) {
8    this.radioProgram = radioProgram;
9    makeUi(); //build UI with Swing Componentes
10  }
11  ...
12 }
```

Finally, we can modify the implementation of the **radio.ui**.*allCompetitions*() and **radio.ui**.*saveInscription*() methods, as follows:

Using Business Logic Services from the UI Layer

```
package radio.ui;

public class InscriptionView {
 ...
 private void allCompetitions() {
  var competitions = new ArrayList<Item>();
  competitions.add(new Item(0, "Select One"));
  for (RadioCompetition comp : radioProgram.availableCompetitions()) {
   competitions.add(new Item(comp.id(), comp.description()));
  }
  this.comboBox = new JComboBox<Item>(new CompetitionComboBoxModel(
         competitions.toArray(new Item[competitions.size()])));
 }

 private void saveInscription() {
  radioProgram.addInscription(
       ((Item) this.comboBox.getSelectedItem()).id(),
         new Competitor() {
          @Override
          public String phone() {
           return txtPhone.getText();
          }
          @Override
          public String name() {
           return txtName.getText();
          }
          @Override
          public String lastName() {
           return txtLastName.getText();
          }
          @Override
          public String id() {
           return txtId.getText();
          }
          @Override
          public String email() {
           return txtEmail.getText();
          }
        });
 }
 ...
}
```

radio.ui.*allCompetitions*() simply invokes **radio.model.api**.*radioProgram.availableCompetitions*() (line 8) to build the list box with the competitions and **radio.ui**.*saveInscription*() creates an instance of **radio.model.api**.*Competitor* with the data entered by the end user. Then invoke **radio.model.api**.*radioProgram.addInscription*() (line 16). Note that the classes of the **radio.ui** package are coupled to interfaces of the **radio.model.api** package, and not to concrete classes or other packages.

Now let's move on to the persistence layer. In order to implement the **radio.model.api**.*RadioProgram* interface, I need to consume services from the persistence layer, exposed via the following interfaces in the **radio.persistence.api** package.

Persistence Layer Public Interfaces

```
1  package radio.persistence.api;
2
3  public interface CompetitionRepository {
4   Optional<Competition> competitionBy(int id);
5   void addInscription(String name, String lastName, String id,
6      String phone, String email, int idCompetition)
7       throws PersistenceException;
8    List<Competition> competitionsForInscription()
9       throws PersistenceException;
10 }
11
12 public interface Competition {
13  int id();
14  String description();
15  String rules();
16  LocalDateTime startDate();
17  LocalDateTime inscriptionStartDate();
18  LocalDateTime inscriptionEndDate();
19 }
```

Next we implement the interface **radio.persistence.api**.*CompetitionRepository*:

Persistence Layer Implementation

```
package radio.persistence;

public class JdbcCompetitorRepository implements CompetitionRepository {
 private String user;
 private String pwd;
 private String dbName;
 private String dbServer;
 private String dbPort;

 public JdbcCompetitorRepository(String user, String pwd, String dbName,
   String dbServer, String dbPort) {
  this.user = user;
  this.pwd = pwd;
  this.dbName = dbName;
  this.dbServer = dbServer;
  this.dbPort = dbPort;
 }

 @Override
 public Optional<Competition> competitionBy(int id) {
  Connection c = connection();
  try {
   var checkCompetitionSt = c
     .prepareStatement("select id, description, rules"
       + ",start_date, inscription_start_date"
       + ",inscription_end_date from competition " + "where id = ?");

   checkCompetitionSt.setInt(1, id);
   ResultSet resultSet = checkCompetitionSt.executeQuery();
   if (resultSet.next()) {
    return Optional.of(toCompetition(resultSet));
   }
   return Optional.empty();
  } catch (SQLException e) {
   throw new PersistenceException(e);
  } finally {
   try {
    c.close();
   } catch (SQLException e) {
    throw new PersistenceException(e);
   }
  }
```

```
43  }
44
45  @Override
46  public void addInscription(String name, String lastName, String id,
47    String phone, String email, int idCompetition)
48    throws PersistenceException {
49   Connection c = connection();
50   try {
51    c.setAutoCommit(false);
52    var st = c.prepareStatement(
53      "insert into competitor(first_name, last_name, person_id, "
54        + "email, phone) " + "values(?,?,?,?,?)",
55      Statement.RETURN_GENERATED_KEYS);
56
57    st.setString(1, name);
58    st.setString(2, lastName);
59    st.setString(3, id);
60    st.setString(4, email);
61    st.setString(5, phone);
62    st.executeUpdate();
63
64    ResultSet generatedKeys = st.getGeneratedKeys();
65    generatedKeys.next();
66
67    PreparedStatement st2 = c.prepareStatement(
68      "insert into inscription(id_competition, id_competitor, inscription_date)"
69        + " values(?,?,?)");
70
71    st2.setInt(1, idCompetition);
72    st2.setInt(2, generatedKeys.getInt(1));
73    st2.setTimestamp(3, Timestamp.valueOf(LocalDateTime.now()));
74    st2.executeUpdate();
75    c.commit();
76   } catch (Exception e) {
77    try {
78     c.rollback();
79     throw new PersistenceException(e);
80    } catch (SQLException s) {
81     throw new PersistenceException(s);
82    }
83   } finally {
84    try {
85     c.setAutoCommit(true);
```

```
86      } catch (SQLException s) {
87       throw new PersistenceException(s);
88      }
89     }
90    }
91
92    @Override
93    public List<Competition> competitionsForInscription()
94      throws PersistenceException {
95     Connection c = connection();
96     try {
97      PreparedStatement st = c
98        .prepareStatement("select id, description, rules, start_date, "
99          + "inscription_start_date, inscription_end_date "
100         + "from competition " + "where inscription_start_date <= ? "
101         + "and inscription_end_date >= ?");
102
103     st.setTimestamp(1, Timestamp.valueOf(LocalDateTime.now()));
104     st.setTimestamp(2, Timestamp.valueOf(LocalDateTime.now()));
105
106     ResultSet resultSet = st.executeQuery();
107     var competitions = new ArrayList<Competition>();
108     while (resultSet.next()) {
109      competitions.add(toCompetition(resultSet));
110     }
111     return competitions;
112    } catch (SQLException e) {
113     throw new PersistenceException(e);
114    } finally {
115     try {
116      c.close();
117     } catch (SQLException e) {
118      throw new PersistenceException(e);
119     }
120    }
121   }
122
123   private Competition toCompetition(ResultSet resultSet) {
124    try {
125     int id = resultSet.getInt("id");
126     var description = resultSet.getString("description");
127     var rules = resultSet.getString("rules");
128     var startDate = resultSet.getTimestamp("start_date");
```

```
129     var inscriptionStartDate = resultSet
130       .getTimestamp("inscription_start_date");
131     var inscriptionEndDate = resultSet.getTimestamp("inscription_end_date");
132
133     return new Competition() {
134      @Override
135      public LocalDateTime startDate() {
136       return startDate.toLocalDateTime();
137      }
138      @Override
139      public String rules() {
140       return rules;
141      }
142      @Override
143      public LocalDateTime inscriptionStartDate() {
144       return inscriptionStartDate.toLocalDateTime();
145      }
146      @Override
147      public LocalDateTime inscriptionEndDate() {
148       return inscriptionEndDate.toLocalDateTime();
149      }
150      @Override
151      public int id() {
152       return id;
153      }
154      @Override
155      public String description() {
156       return description;
157      }
158     };
159    } catch (SQLException e) {
160     throw new PersistenceException(e);
161    }
162   }
163  }
```

From the source code above, first note that the class name is prefixed with *Jdbc* because it is the technology we use for the implementation. On the other hand, the data required to connect to the database is now injected by constructor. Also, note that we are catching the *SQLException* and throwing an exception that belongs to the persistence layer. *SQLException* reveals the use of a relational database, if it escapes from the persistence layer, reaching the Business or UI layer, we would be filtering technical aspects where it does not correspond. Which would have an impact if, for

example, we wanted to use a NoSQL database in the future. Finally note that the implementation of the interface **radio.persistence.api**.*Competition* is done inside the private method *toCompetition*() (line 123).

Lets implement now the interfaces defined in the **radio.model.api** package:

Business Layer Implementation

```
1  package radio.model;
2
3  public class DefaultRadioProgram implements RadioProgram {
4   private CompetitionRepository repository;
5   public DefaultRadioProgram(CompetitionRepository repository) {
6    this.repository = repository;
7   }
8   @Override
9   public Iterable<RadioCompetition> availableCompetitions() {
10   List<Competition> cs = repository.competitionsForInscription();
11   return cs.stream().map(d -> new RadioCompetition() {
12    @Override
13    public int id() {
14     return d.id();
15    }
16    @Override
17    public String description() {
18     return d.description();
19    }
20   }).collect(Collectors.toList());
21  }
22  @Override
23  public void addInscription(int idCompetition, Competitor competitor) {
24   repository.competitionBy(idCompetition)
25        .orElseThrow(() -> new RadioException(
26          "Selected competition does not exists..."));
27
28   Competitor c = new DefaultCompetitor(competitor.id(),
29        competitor.name(), competitor.lastName(),
30        competitor.email(), competitor.phone());
31
32   repository.addInscription(c.name(), c.lastName(), c.id(),
33        c.phone(), c.email(), idCompetition);
34  }
35 }
36 class DefaultCompetitor implements Competitor {
37  private String id;
```

```
38   private String name;
39   private String lastName;
40   private Email email;
41   private Phone phone;
42
43   public DefaultCompetitor(String id, String name, String lastName,
44       String email, String phone) {
45    this.id = new NotNullNorEmpty<String>(id, "Id").value();
46    this.name = new NotNullNorEmpty<String>(name, "Name").value();
47    this.lastName = new NotNullNorEmpty<String>(lastName, "Last Name").value();
48    this.email = new Email(email);
49    this.phone = new Phone(phone);
50   }
51   public String id() {
52    return id;
53   }
54   public String name() {
55    return name;
56   }
57   public String lastName() {
58    return lastName;
59   }
60   public String email() {
61    return email.toString();
62   }
63   public String phone() {
64    return phone.toString();
65   }
66  }
```

The **radio.model**.*DefaultRadioProgram* implementation does some business validations on the data that comes from the *ui* layer (lines 45 - 49) and delegates to the services exposed by the *persistence* layer. Again, the *business* layer is coupled to the interface **radio.persistence.api**.*CompetitionRepository* as figure 3.5 depicted. Layers are always coupled through interfaces, the most abstract syntactical construction that a programming language provides.

Below are the business validations that were moved from the *ui* layer to the *business* layer.

Business Validations

```
package radio.model;

class Email {
 private String email;
 private static String REGEX = "^[\\w-_\\.+]*[\\w-_\\.]\\@([\\w]+\\.)+[\\w]+[\\w]$";
 public Email(String email) {
  if (!email.matches(REGEX)) {
   throw new RadioException("Must be a valid email address...");
  }
  this.email = email;
 }
 @Override
 public String toString() {
  return email;
 }
}
class Phone {
 public String number;
 private static String REGEX = "\\d{4}-\\d{6}";
 public Phone(String number) {
  if (!number.matches(REGEX)) {
   throw new RadioException("Must be a valid phone number NNNN-NNNNNN");
  }
  this.number = number;
 }
 @Override
 public String toString() {
  return number;
 }
}
```

These validations must be in the *business* layer due to they are business rules, but in addition, if we keep them in the *ui* layer when implementing another user interface (web for example), we should duplicate those validations there. These validations throw the *RadioException* and the *ui* layer catch it showing the error to the end user as below:

UI Layer Showing Business Exceptions

```
package radio.ui;

public class InscriptionView {
 ...

  btnOk.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    var worker = new SwingWorker<Void, Void>() {
     @Override
      protected void done() {
       try {
        btnOk.setEnabled(true);
        get();
        JOptionPane.showMessageDialog(contentPane,
           "Inscription done successfully !");
       } catch (ExecutionException e) {
        if (RadioException.class.equals(e.getCause().getClass())) {
         JOptionPane.showMessageDialog(contentPane, e.getMessage());
        }
        else {
         JOptionPane.showMessageDialog(contentPane, "Something went wrong...");
         throw new RuntimeException(e);
        }
       } catch (InterruptedException e) {
        JOptionPane.showMessageDialog(contentPane, "Something went wrong...");
        throw new RuntimeException(e);
       }
      }
      @Override
      protected Void doInBackground() throws Exception {
       btnOk.setEnabled(false);
       saveInscription();
       // to make the compiler happy...
       return null;
      }
    };
    worker.execute();
   }
  });
 ...
}
```

Finally we have the *Main* program, who instantiate the application with the specific implementations of each layer:

Main Program

```
package main;

public class Main {
 public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
   @Override
   public void run() {
    try {
     new Main().start();
    } catch (Exception e) {
     // log the exception...
     //Show "Ups something went wrong" to the user...
    }
   }
  });
 }

 private void start() {
  new InscriptionView(
    new DefaultRadioProgram(
      new JdbcCompetitorRepository(
        "app", "xxxxx", "radiocompetition", "localhost", "1527")
      )
    );
 }
}
```

Figure 3.6 shows the layered structure of "RadioA" as we have coded it, including the dependencies between the most important classes and interfaces we have defined.

Next lets add the corresponding validation of this architecture using ArchUnit*:

*https://www.archunit.org/

Testing the Layered Architecture

```
public class TestArchitecture {

 private static final String PKG_UI = "radio.ui";
 private static final String PKG_MODEL_API = "radio.model.api";
 private static final String PKG_MODEL_IMPL = "radio.model";
 private static final String PKG_PERSISTENCE_API = "radio.persistence.api";
 private static final String PKG_PERSISTENCE_IMPL = "radio.persistence";
 private static final String ROOT_PACKAGE = "radio";

 @Test
 public void layersAreValid() {
  layeredArchitecture()
    .layer("UI").definedBy(PKG_UI)
    .layer("BusinessLogicAPI").definedBy(PKG_MODEL_API)
    .layer("BusinessLogicImpl").definedBy(PKG_MODEL_IMPL)
    .layer("PersistenceAPI").definedBy(PKG_PERSISTENCE_API)
    .layer("PersistenceImpl").definedBy(PKG_PERSISTENCE_IMPL)
    .whereLayer("UI").mayNotBeAccessedByAnyLayer()
    .whereLayer("BusinessLogicImpl").mayNotBeAccessedByAnyLayer()
    .whereLayer("BusinessLogicAPI").mayOnlyBeAccessedByLayers("UI", "BusinessLogicIm\
pl")
    .whereLayer("PersistenceAPI").mayOnlyBeAccessedByLayers("BusinessLogicImpl", "Pe\
rsistenceImpl")
    .whereLayer("PersistenceImpl").mayNotBeAccessedByAnyLayer()
    .check(new ClassFileImporter().importPackages(ROOT_PACKAGE));
 }

 @Test
 public void uiShouldOnlyDependOnModelApiOrJdk() {
  JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);

  ArchRule r1 = classes().that().resideInAPackage(PKG_UI).should()
    .onlyDependOnClassesThat()
    .resideInAnyPackage(PKG_UI, PKG_MODEL_API, "java..", "javax..");
  r1.check(jc);
 }

 @Test
 public void modelAPIShouldNotDependOnOtherPackages() {
  JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);

  ArchRule r1 = classes().that().resideInAPackage(PKG_MODEL_API)
```

```
43      .should().onlyDependOnClassesThat()
44      .resideInAnyPackage(PKG_MODEL_API, "java..");
45    r1.check(jc);
46   }
47
48   @Test
49   public void modelShouldOnlyDependOnModelOrJdk() {
50    JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
51
52    ArchRule r1 = classes().that().resideInAPackage(PKG_MODEL_IMPL).should()
53      .onlyDependOnClassesThat().resideInAnyPackage(PKG_MODEL_IMPL,
54        PKG_MODEL_API, PKG_PERSISTENCE_API, "java..");
55    r1.check(jc);
56   }
57
58   @Test
59   public void persistenceAPIShouldNotDependOnOtherPackages() {
60    JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
61
62    ArchRule r1 = classes().that().resideInAPackage(PKG_PERSISTENCE_API)
63      .should().onlyDependOnClassesThat()
64      .resideInAnyPackage(PKG_PERSISTENCE_API, "java..");
65    r1.check(jc);
66   }
67
68
69   @Test
70   public void persistenceShouldOnlyDependOnPersistenceOrJdk() {
71    JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
72
73    ArchRule r1 = classes().that().resideInAPackage(PKG_PERSISTENCE_IMPL)
74      .should().onlyDependOnClassesThat()
75      .resideInAnyPackage(PKG_PERSISTENCE_IMPL, PKG_PERSISTENCE_API, "java..");
76    r1.check(jc);
77   }
78  }
```

With this test we make sure that developers by mistake does not generate dependencies between classes of layers that are not allowed.

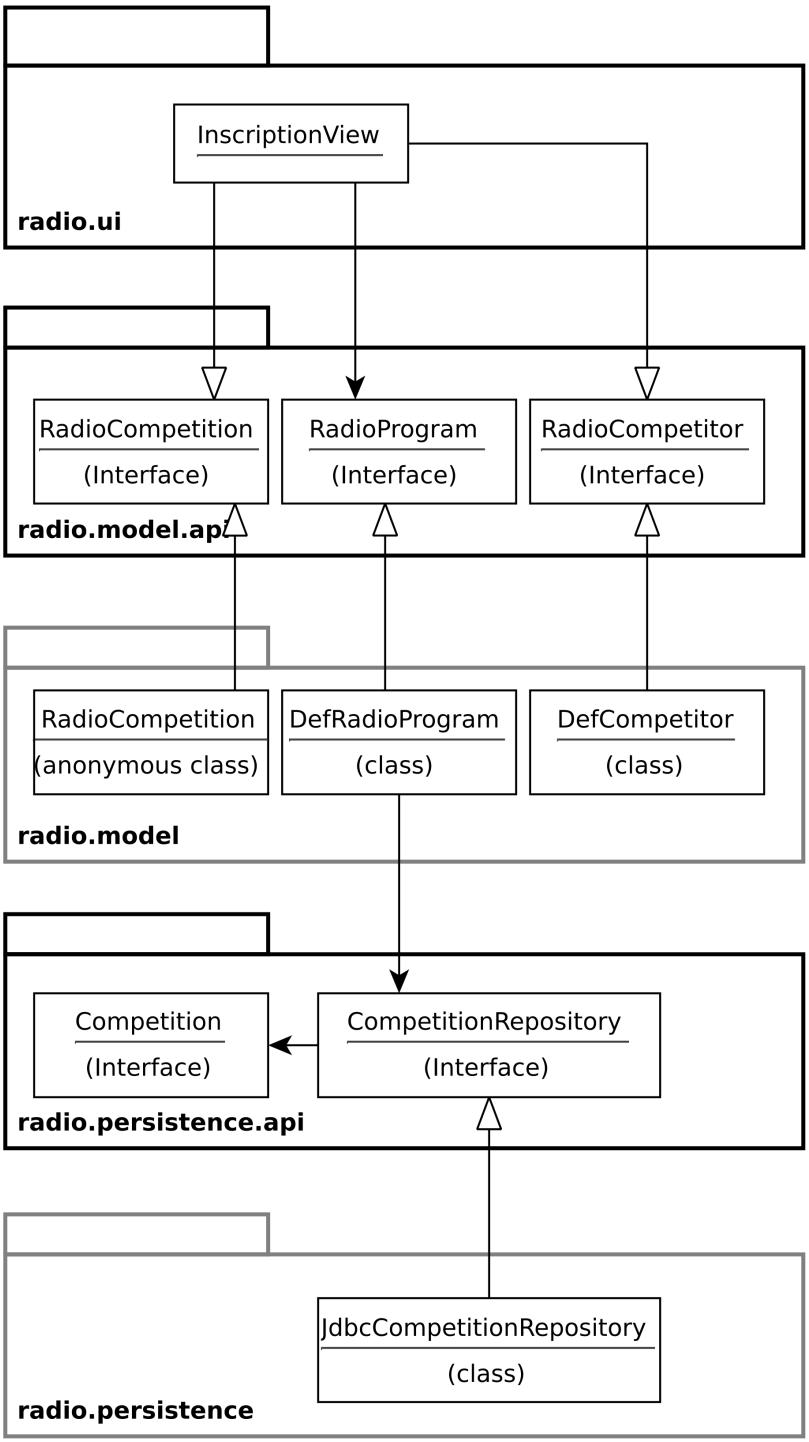

Figure 3.6: Layered with main classes

Full source code available at github/radioa-layers*.

*https://github.com/enriquemolinari/radioa-layers

Hexagonal Architecture

Definitions and Style

In 2005 Alistair Cockburn published the architecture style called Hexagonal Architecture*, also known as **Ports** and **Adapters**. The intention of this style as defined by its author says:

"Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases".

It is important to note that this style has special emphasis on the rule of separating the application (business logic) from any external element such as user interfaces, scripts, email services, external applications, messaging systems, databases, etc.

To achieve this separation (decoupling) the application communicates only through interfaces - called **ports** - with external infrastructure elements. We have two flavors of ports, those who drive the application (or primary ports) and those who the application drives (secondary ports). So, we have external elements such as scripts, user interfaces, automated tests, etc., who drive the application, which means that invokes the API that the application exposes via the ports. And we have, on the other hand, the application communicating via the ports with external elements such as databases, external systems, etc by invoking secondary ports. In both cases, for each external element there will be one or more **adapters** that will be responsible for the mapping between the outside world and the application (inside) world.

Figure 3.7 shows the typical diagram of the hexagonal architecture:

*https://alistair.cockburn.us/hexagonal-architecture/

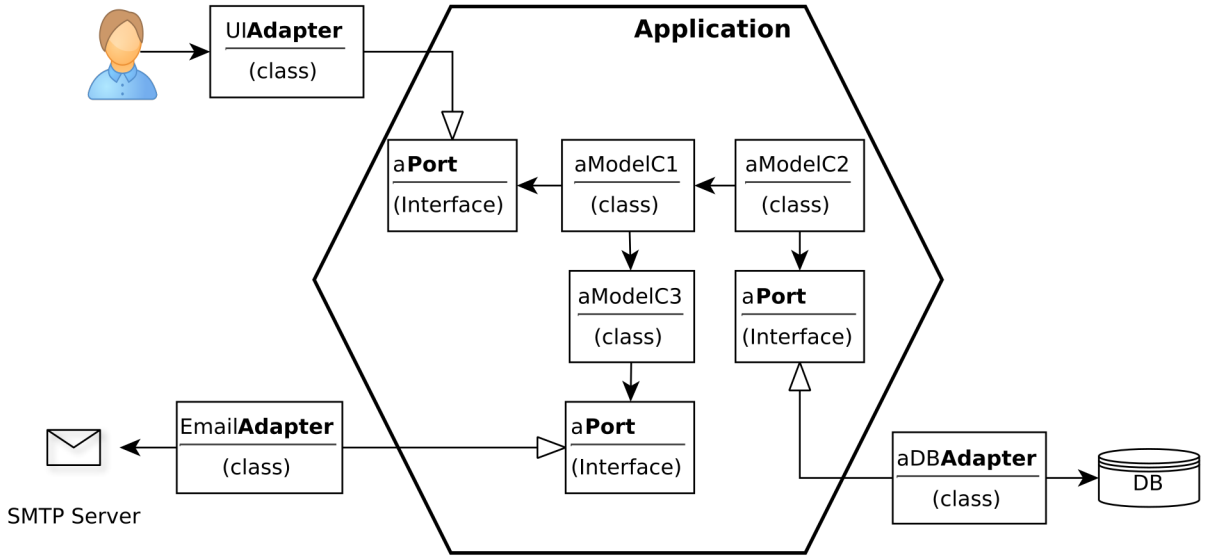

Figure 3.7: Hexagonal Architecture

As can be seen, within the hexagon are the classes that implement the business logic, which in this style is just called *application*. The *aPort* interfaces inside the hexagon, called **ports**, are the input connection to the application (primary ports) and the output connection to external elements (secondary ports). For inputs, the application exposes a set of APIs via interfaces (ports) that are consumed from the outside. In this case it is the outsides who drives the application. For output, the application exposes SPI (Service Provider Interface) interfaces (ports), and the external elements implement them to provide the service invoked (driven) by the application.

The adapters belong to the outside of the application and are those that are coupled to technical aspects (frameworks, drivers, external services, etc). We could have several different adapters for the same port.

This architecture style forces us to put our efforts on functional requirements from the very beginning. Go further implementing business logic, to later decide which frameworks and external services to use. It also allows us to fully test the application with unit tests, without mocking frameworks, just simple classes in our tests will implement the ports that allow us to exercise the different use cases.

In 2012 Robert C. Martin publish a blog post that made the term Clean Architecture (Clean Architecture*) known, promoting these architectural styles that prioritize applications free of frameworks, database drivers, UI technologies and that allow to be tested isolated from any technical aspect.

Implementation

If we look at the layered structure that we defined for the "RadioA" application (figures 3.5 or 3.6), in the previous section, we can see that the business logic, which is located in the **radio.model** package,

*https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

needs the persistence layer, its interfaces from the **radio**.**persistence**.**api** package in order to work (even compiled if we are using a compiled language). Similarly, it is not possible to test the business layer without including the interfaces from the persistence layer.

Therefore, if we want to refactor RadioA's layered architecture and turn it into a hexagonal architecture we must **invert the dependency** that exists between **radio**.**model** and **radio**.**persistence**. Instead of having the **radio**.**model** package depending on the **radio**.**persistence** package, we should make the **radio**.**persistence** package depend on the **radio**.**model** package. To do that, we make the following changes:

1. We create the package **radio**.**model**.**ports** and move there the interfaces of the packages **radio**.**model**.**api** and **radio**.**persistence**.**api**. Then we remove these *.api* packages.
2. We **remove** the interface **radio**.**persitence**.*Competition*, since we will use **model**.**ports**.*RadioCompetition*.
3. We rename the packages *radio.ui* and *radio.persistence* to *radio*.**adapters**.*ui* and *radio*.**adapters**.*persistence*.
4. We **modify** the class **radio**.**persistence**.*JdbcCompetitionRepository* to return an instance of **radio**.**model**.**ports**.*RadioCompetition*.

With these changes, we transform the layered architecture illustrated in figure 3.6 into an hexagonal architecture shown in figure 3.8:

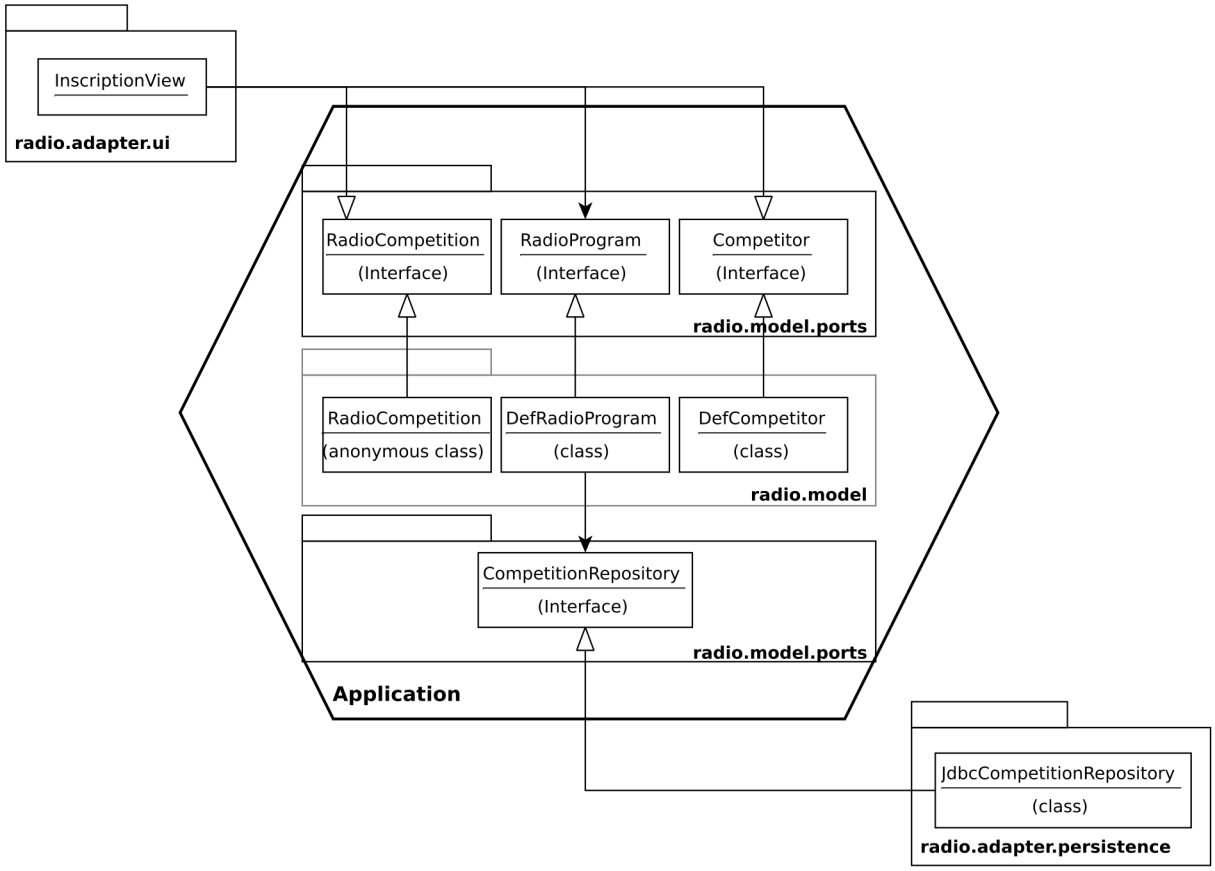

Figure 3.8: RadioA, Hexagonal Architecture

Note that the direction of the dependencies goes from outside the hexagon towards the **radio.model.ports** package inside the hexagon.

Lets then write a test to validate the rules imposed by the hexagonal architecture:

Testing the Hexagonal Architecture

```
1  public class TestArchitecture {
2
3   private static final String ROOT_PACKAGE = "radio";
4   private static final String PKG_MODEL = "radio.model";
5   private static final String PKG_PORTS = "radio.model.ports";
6   private static final String PKG_ADAPTER = "radio.adapter..";
7
8   @Test
9   public void testLayers() {
10   layeredArchitecture()
11     .layer("Adapter").definedBy(PKG_ADAPTER)
12     .layer("Ports").definedBy(PKG_PORTS)
```

```
13      .layer("BusinessLogic").definedBy(PKG_MODEL)
14      .whereLayer("Adapter").mayNotBeAccessedByAnyLayer()
15      .whereLayer("BusinessLogic").mayOnlyBeAccessedByLayers("Ports")
16      .check(new ClassFileImporter().importPackages(ROOT_PACKAGE));
17    }
18
19    @Test
20    public void modelShouldOnlyDependOnPortsOrJdk() {
21     JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
22
23     ArchRule r1 = classes().that().resideInAPackage(PKG_MODEL).should()
24       .onlyDependOnClassesThat()
25       .resideInAnyPackage(PKG_MODEL, PKG_PORTS, "java..");
26     r1.check(jc);
27    }
28
29    @Test
30    public void portsShouldOnlyDependOnPortsOrJdk() {
31     JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
32
33     ArchRule r1 = classes().that().resideInAPackage(PKG_PORTS).should()
34       .onlyDependOnClassesThat()
35       .resideInAnyPackage(PKG_PORTS, "java..");
36     r1.check(jc);
37    }
38
39    @Test
40    public void adaptersShouldOnlyDependOnPorts() {
41     JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
42
43     ArchRule r1 = classes().that().resideInAPackage(PKG_ADAPTER).should()
44       .onlyDependOnClassesThat()
45       .resideInAnyPackage(PKG_ADAPTER, PKG_PORTS, "java..", "javax..");
46     r1.check(jc);
47    }
48
49    @Test
50    public void adaptersShouldNotDependOnEachOther() {
51     JavaClasses jc = new ClassFileImporter().importPackages(ROOT_PACKAGE);
52
53     SliceRule rule = SlicesRuleDefinition.slices()
54        .matching(PKG_ADAPTER + "(**)")
55        .should().notDependOnEachOther();
```

```
56     rule.check(jc);
57   }
58 }
```

As an example, I will also add an Ant script to verify (again) the architecture rules. The Ant script compiles and generates the artifacts verifying the rules imposed by the hexagonal architecture.

Verifying the Hexagonal Architecture Rules with Ant

```
1  <!-- compiling application only -->
2
3  <target name="compile-ports">
4   <mkdir dir="physical-build/ports" />
5   <javac sourcepath="" srcdir="src/main/java"
6     destdir="physical-build/ports"
7     excludes="radio/model/*.java"
8     includes="radio/model/ports/*.java">
9   </javac>
10 </target>
11
12 <target name="jar-ports" depends="compile-ports">
13  <mkdir dir="physical-build/jar-ports"/>
14
15  <jar destfile="physical-build/jar-ports/ports.jar"
16     basedir="physical-build/ports">
17  </jar>
18 </target>
19
20 <target name="compile-application" depends="jar-ports">
21  <mkdir dir="physical-build/classes-application" />
22
23  <javac sourcepath="" srcdir="src/main/java"
24    classpath="physical-build/jar-ports/ports.jar"
25    destdir="physical-build/classes-application"
26    excludes="radio/model/ports/*.java" includes="radio/model/*.java">
27  </javac>
28 </target>
29
30 <target name="jar-application" depends="compile-application">
31  <mkdir dir="physical-build/jar-application"/>
32
33  <jar destfile="physical-build/jar-application/application.jar"
34      basedir="physical-build/classes-application">
35  </jar>
```

```
</target>

<!-- compile infrastructure depending on the ports artifact only -->

<target name="compile-persistence" depends="jar-ports">

 <mkdir dir="physical-build/classes-persistence" />

 <javac sourcepath=""
   classpath="physical-build/jar-ports/ports.jar"
   srcdir="src/main/java"
   destdir="physical-build/classes-persistence"
   includes="radio/adapter/persistence/*.java">
 </javac>
</target>

<target name="jar-persistence" depends="compile-persistence">
 <mkdir dir="physical-build/jar-persistence"/>

 <jar destfile="physical-build/jar-persistence/persistence.jar"
   basedir="physical-build/classes-persistence">
 </jar>
</target>

<target name="compile-ui" depends="jar-ports">
 <mkdir dir="physical-build/classes-ui" />
  <javac
    sourcepath="" srcdir="src/main/java"
    classpath="physical-build/jar-ports/ports.jar"
    destdir="physical-build/classes-ui"
    includes="radio/adapter/ui/**/*.java">
  </javac>
</target>

<target name="jar-ui" depends="compile-ui">
 <mkdir dir="physical-build/jar-ui"/>

 <jar destfile="physical-build/jar-ui/ui.jar"
      basedir="physical-build/jar-ui">
 </jar>
</target>
```

The Ant script compiles first the ports package and generates a *jar* with its classes. This *jar* is then used as the only dependency on all the other packages. A *jar* is also generated for each of the other packages. This is illustrated in Figure 3.9.

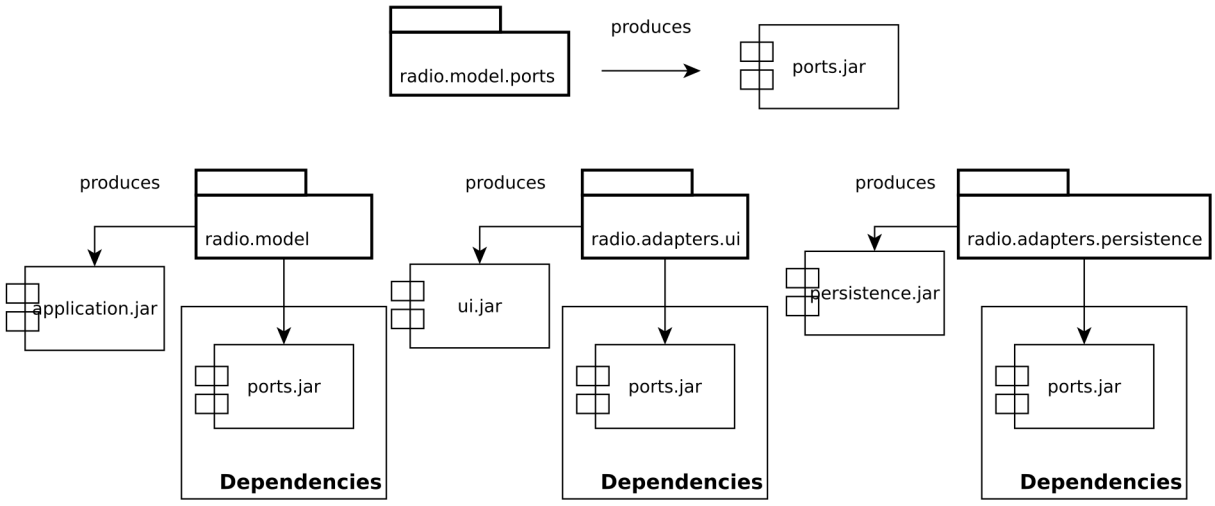

Figure 3.9: Separated Compilation with Ant

As you can see, each package just needs the *ports.jar* to compile as required by the hexagonal architecture. Executing these validations on every commit keeps us aligned with the architecture rules of the hexagonal style which give us clarity about where the changes will impact.

The source code of this implementation can be found in github/radioa-hexagonal*.

Inverted Layered Architecture

Style and Implementation

Having then understood the ideas of the hexagonal architecture let's look at another variant. We are going to implement RadioA by modeling the requirements in objects. What Fowler calls the rich domain model[34] [35] (**not anemic**[36]). And to persist the object model we are going to use a Data Mapper†[37] solution like JPA (Java Persistence API), which allows us to enjoy the benefits of transparent persistence[38] [39].

So, let's briefly review the requirements we had:

1. Registrations in competitions are only valid between valid inscription dates.
2. **New**: Those participants who register on the first day of registration will receive ten points, which can then be exchanged later for different gifts.

*https://github.com/enriquemolinari/radioa-hexagonal
†https://martinfowler.com/eaaCatalog/dataMapper.html

In future iterations we will add the possibility to listeners to authenticate and do the inscription in competitions by themselves. For now, an employee of the radio is still the end user of the application, filling all the data in the form shown in the figure 3.3.

To implement the requirements we define the following classes (with their most important responsibilities) and relationships:

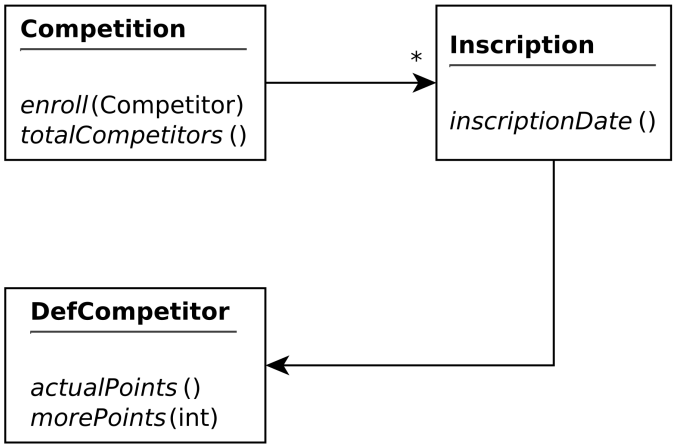

Figure 3.10: RadioA, Domain Model

Let's see the source code of that domain model:

Domain Model

```
1  package radio.model
2
3  public class DefaultCompetitor implements Competitor {
4   ...
5   public void morePoints(int points) {
6    if (points <= 0) {
7     throw new RadioException("points must be positive number...");
8    }
9    this.points += points;
10  }
11
12  public int actualPoints() {
13   return points;
14  }
15   ...
16 }
17
18 public class Competition implements RadioCompetition {
19  private List<Inscription> inscriptions;
```

```
20   ...
21   public Competitor enroll(Competitor competitor) {
22    var inscriptionDate = LocalDateTime.now();
23    if (!inTime(inscriptionDate))
24     throw new RadioException("Out of inscription date...");
25
26    var dCompetitor = DefaultCompetitor.of(competitor);
27
28    if (sameDayAsInscriptionStart(inscriptionDate))
29     dCompetitor.morePoints(10);
30
31    this.inscriptions.add(new Inscription(dCompetitor, inscriptionDate));
32
33    return dCompetitor;
34   }
35
36   public int totalCompetitors() {
37    return inscriptions.size();
38   }
39   ...
40  }
41
42  public class Inscription {
43   private DefaultCompetitor competitor;
44   private LocalDateTime date;
45   ...
46   public String inscriptionDate() {
47    return date.toString();
48   }
49   ...
50  }
```

With these three classes I can implement the requirements defined above, and I can also test them completely isolated from the other layers, as shown below:

Unit Testing Domain Model

```
public class CompetitionTest {
 @Test
 public void enroll_competitor() {
  var competition = new Competition(1, "competiton 1", "rules...",
    LocalDateTime.now().plusDays(20), LocalDateTime.now().minusDays(1),
    LocalDateTime.now().plusDays(5));

  var competitor = competition.enroll(new DefaultCompetitor("1234",
    "Enrique", "Molinari", "enrique.molinari@gmail.com", "1234-987654"));

  Assert.assertTrue(competition.competitor("1234"));
  Assert.assertEquals(0, competitor.actualPoints());
 }

 @Test
 public void enroll_competitor_same_day_inscription_start() {
  var competition = new Competition(1, "competiton 1", "rules...",
    LocalDateTime.now().plusDays(20), LocalDateTime.now(),
    LocalDateTime.now().plusDays(5));

  var competitor = competition.enroll(new DefaultCompetitor("1234",
    "Enrique", "Molinari", "enrique.molinari@gmail.com", "1234-987654"));

  Assert.assertEquals(10, competitor.actualPoints());
 }

 @Test(expected = RadioException.class)
 public void cant_enroll() {
  var competition = new Competition(1, "competiton 1", "rules...",
    LocalDateTime.now().plusDays(20), LocalDateTime.now().plusDays(1),
    LocalDateTime.now().plusDays(5));

  competition.enroll(new DefaultCompetitor("1234", "Enrique", "Molinari",
    "enrique.molinari@gmail.com", "1234-987654"));

  Assert.assertTrue(false);
 }
}
```

Then we have the API (interfaces) exposed by the application so that the package **radio.ui** (or any other client) can invoke. These API does not change with respect to what was implemented in

the hexagonal architecture, they are the same. SPI interfaces do require changes, to allow us to take advantage of the DataMapper (ORM), such as transparent persistence, lazy loading, etc. Let us remember that these advantages to work requires what some DataMappers/ORMs like Hibernate* call persistence context†. For this I need an abstract mechanism (interface) that allows me to define the limits where I am going to use my persistent objects. We then define the following two interfaces within the **radio.model.api** package:

SPI Interfaces

```
1  package radio.model.api;
2
3  public interface UnitOfWork {
4   <R> R tx(Function<Competitions, R> codeInTx);
5  }
6
7  public interface Competitions {
8   Optional<Competition> competitionBy(int id);
9
10  List<Competition> competitionsForInscription();
11 }
```

radio.model.api.*UnitOfWork.tx* allows me to pass by parameter (*codeInTx*) code blocks that manipulate persistent objects, executing them within a persistence context. We can do this using lambda expressions that implement *java.util.function.Function*. Then **radio.model.api**.*Competitions* is the SPI interface that will allow me to get competitions in two different ways from the persistent repository.

So now let's see how we implement the **radio.model.api**.*RadioProgram* interface, to provide services to the **radio.ui** layer to obtain competitions and register participants.

Implementing RadioProgram Interface

```
1  package radio.model;
2
3  public class DefaultRadioProgram implements RadioProgram {
4   private UnitOfWork unit;
5   public DefaultRadioProgram(UnitOfWork unit) {
6    this.unit = unit;
7   }
8   @Override
9   public Iterable<RadioCompetition> availableCompetitions() {
10   Function<Competitions, List<Competition>> codeBlock = (competitions) -> {
11    return competitions.competitionsForInscription();
```

*https://docs.jboss.org/hibernate/orm/5.0/devguide/en-US/html/index.html
†https://docs.jboss.org/hibernate/orm/5.0/devguide/en-US/html/ch03.html

```
12    };
13    var competitions = unit.tx(codeBlock);
14    return competitions.stream().map(d -> new RadioCompetition() {
15     @Override
16     public int id() {
17      return d.id();
18     }
19     @Override
20     public String description() {
21      return d.description();
22     }
23     @Override
24     public String rules() {
25      return d.rules();
26     }
27     @Override
28     public LocalDateTime startDate() {
29      return d.startDate();
30     }
31     @Override
32     public LocalDateTime inscriptionStartDate() {
33      return d.inscriptionStartDate();
34     }
35     @Override
36     public LocalDateTime inscriptionEndDate() {
37      return d.inscriptionEndDate();
38     }
39    }).collect(Collectors.toList());
40   }
41
42   @Override
43   public void addInscription(int idCompetition, Competitor competitor) {
44    unit.tx((competitions) -> {
45     var competition = competitions.competitionBy(idCompetition).orElseThrow(
46       () -> new RadioException("Selected competition does not exists..."));
47
48     Competitor c = new DefaultCompetitor(competitor.personId(),
49       competitor.name(), competitor.lastName(), competitor.email(),
50       competitor.phone());
51
52     competition.enroll(c);
53    });
54   }
```

```
55 }
```

Let's take a moment to look at the implementation of *RadioProgram.addInscription*. Note that everything runs within *UnitOfWork.tx*, that is, within the persistence context. The invocation of *competition.enroll* will not only persistently register a new participant to the competition, but will also persistently assign ten points to the participant if she registers on the first day of registration. Note once again that the same unit test logic above is being used here in production code, only when invoked within the *persistence context*, objects will be persisted without the need for an *update* or *insert* statement. This is called **transparent persistence**.

Let's now see how this transparent persistence is implemented using JPA by implementing the SPI interfaces that the business layer exposes.

Implementing UnitOfWork and Competitions Interfaces

```
1  package radio.persistence;
2
3  public class JpaCompetitions implements Competitions {
4   private EntityManager em;
5   public JpaCompetitions(EntityManager em) {
6    this.em = em;
7   }
8   @Override
9   public Optional<Competition> competitionBy(int id) {
10   var c = em.find(Competition.class, id);
11   if (c != null)
12    return Optional.of(em.find(Competition.class, id));
13   return Optional.empty();
14  }
15  @Override
16  public List<Competition> competitionsForInscription() {
17   return em
18     .createQuery(
19       "from Competition c where :today "
20         + "between c.inscriptionStartDate and c.inscriptionEndDate",
21       Competition.class)
22     .setParameter("today", LocalDateTime.now()).getResultList();
23  }
24 }
25
26 public class JpaUnitOfWork implements UnitOfWork {
27  private EntityManagerFactory emf;
28  public JpaUnitOfWork(String persistenceUnit) {
29   this.emf = Persistence.createEntityManagerFactory(persistenceUnit);
```

```
30   }
31   @Override
32   public <R> R tx(Function<Competitions, R> codeInTx) {
33    EntityManager em = emf.createEntityManager();
34    EntityTransaction tx = em.getTransaction();
35    try {
36     tx.begin();
37     R result = codeInTx.apply(new JpaCompetitions(em));
38     tx.commit();
39     return result;
40    } catch (RadioException e) {
41     tx.rollback();
42     throw e;
43    } catch (Exception e) {
44     tx.rollback();
45     throw new RuntimeException(e);
46    } finally {
47     em.close();
48    }
49   }
50  }
```

radio.persistence.*JpaUnitOfWork* implements the persistence context using the JPA *EntityManager*. Note that the method *tx*() receives a lambda expression *codeInTx* (generated in the business layer) and executes it via the call to *Function.apply*(), passing it as a parameter an instance of **radio.persistence**.*JpaCompetitions*, who receives the *EntityManager* instance to be able to perform the queries that can be seen in the implementation of **radio.peristence**.*JpaCompetitions*. The use of lambda expressions helps us to have the **try/catch/finally** structure in one place, without duplicating it.

Figure 3.11 shows a layered diagram from what we have implemented:

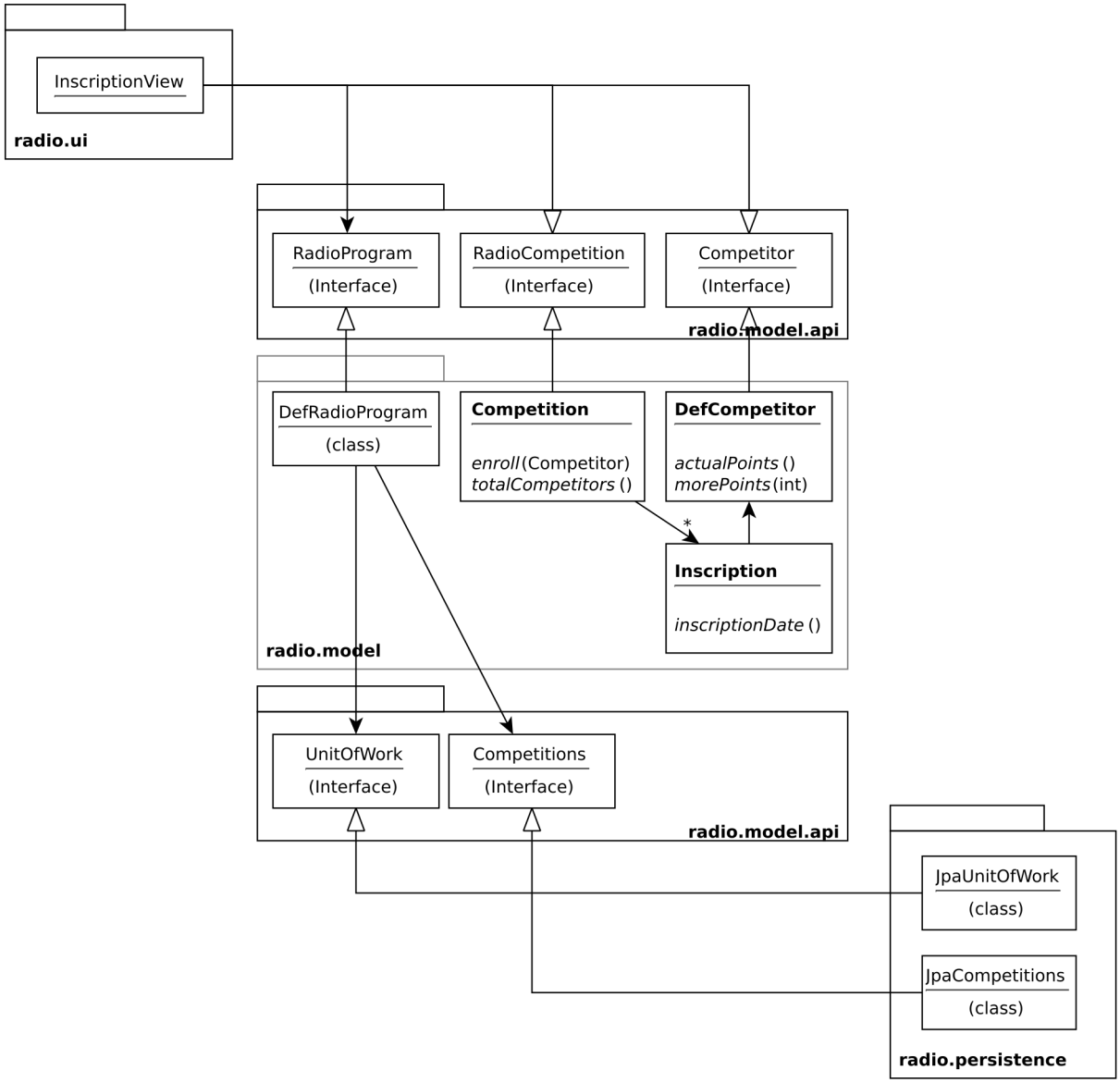

Figure 3.11: RadioA, Hexagonal Architecture

To start the application we have the *Main* program:

Starting the Application

```
public class Main {
 public static void main(String[] args) {
  SwingUtilities.invokeLater(new Runnable() {
   @Override
   public void run() {
    try {
     new Main().start();
    } catch (Exception e) {
     // log the exception...
     // Show "Ups something went wrong" to the user...
    }
   }
  });
 }
 private void start() {
  new InscriptionView(
    new DefaultRadioProgram(
      new JpaUnitOfWork("prod-conn")
    )
  );
 }
}
```

And we test the architecture rules:

Testing Architecture Rules

```
...
private static final String PKG_PERSISTENCE = "radio.persistence";
private static final String PKG_MODEL = "radio.model";
private static final String PKG_MODEL_API = "radio.model.api";
private static final String PKG_UI = "radio.ui";
private static final String ROOT_PKG = "radio";

@Test
public void testLayers() {
 layeredArchitecture()
  .layer("UI").definedBy(PKG_UI)
  .layer("BusinessLogicAPI").definedBy(PKG_MODEL_API)
  .layer("BusinessLogic").definedBy(PKG_MODEL)
  .layer("Persistence").definedBy(PKG_PERSISTENCE)
  .whereLayer("UI").mayNotBeAccessedByAnyLayer()
```

```
16    .whereLayer("BusinessLogic").mayNotBeAccessedByAnyLayer()
17    .whereLayer("BusinessLogicAPI").mayOnlyBeAccessedByLayers("BusinessLogic", "UI", "\
18  Persistence")
19    .whereLayer("Persistence").mayNotBeAccessedByAnyLayer()
20    .check(new ClassFileImporter().importPackages(ROOT_PKG));
21  }
22  ...
```

As you can see in figure 3.11, the diagram is similar to the hexagonal style diagram in figure 3.8. But we have a hidden dependency that is not captured by the verification tests and is not valid according to the hexagonal style. The **radio.persistence** layer is not only coupled to the SPI interfaces defined in **radio.model.api**, but is also, by using reflection, coupled to the concrete class **radio.model**.*Competition*. Let's observe below that when we build a query in JPQL on the **radio.persistence** layer, we are naming the concrete class (line 9) that belongs to the model private package:

Invalid Coupling According to the Hexagonal Style

```
1   package radio.persistence;
2
3   public class JpaCompetitions implements Competitions {
4    ...
5    @Override
6    public List<Competition> competitionsForInscription() {
7     return em
8       .createQuery(
9         "from Competition c where :today "
10          + "between c.inscriptionStartDate and c.inscriptionEndDate",
11        Competition.class)
12      .setParameter("today", LocalDateTime.now()).getResultList();
13   }
14   ---
15  }
```

Let's look at some advantages and disadvantages that we see with this style of architecture. Let's start with the cons:

Additional dependencies: We have to understand that the names of the persistent classes and their private attributes are used from outside of the **radio.model** layer, that is, that I not only have to take care when modifying the interfaces, but also the persistent classes. This is an invalid dependency in the hexagonal architecture style.

Coupling to Frameworks: My application (the **radio.model** layer) although as you noticed it does not have JPA annotations (since the mapping was defined in XML), that is, it is free of frameworks,

it was indirectly coupled to a DataMapper type of framework, because I am making use of the transparent persistence that these frameworks provide. This means that you could not simply switch to any persistence framework other than this type. I could for example move to JDO* and any persistence mechanism that implements JPA or JDO, or transparent persistence. But I can't move without modifying something in the **radio.model** layer, for example, to use plain JDBC or frameworks like JOOQ† or QueryDSL‡.

Continuing with the advantages:

Direction of dependencies: I continue to have dependencies from the outside (infrastructure elements) to the inside (application). For this reason the architecture rule test continues to pass. Which allows me to test my application (**radio.model**) completely isolated as Cockburn describes in his hexagonal architecture.

Transparent persistence: I have the possibility to use the benefits of transparent persistence, lazy loading, delete orphans, etc.

It would be possible to use JPA or similar frameworks in the hexagonal style but we would have to have all the persistent classes inside the persistence layer, and an object model (non-persistent) inside the hexagon. However, I lose the transparent persistence that JPA offers me. This is where we must make a decision and be consistent in its implementation.

Complete source code of this implementation can be found in github/inverted-layered§.

Modular Architecture

Definitions and Style

Splitting an application into modules is a complex design and architecture job that has been studied since the beginning of our profession. What criteria is appropriate to divide an application into modules? In 1971 David Parnas published one of the most recognized works on this subject: On the criteria to be used in decomposing systems into modules[40]. There he mentions that the effectiveness of modularizing depends on the criteria we take to divide an application into modules. That is, depending on the criteria with which we define the modules of an application, we will see to a greater or lesser extent advantages in reducing complexity and improving maintainability.

And what criteria should we use then? It depends a lot on what type of application we are building. It is very different to build a library to be reused in several applications than to build an enterprise application[41]. For example, when building an enterprise application, designing modules that benefit overall system comprehension, reduce complexity, and also allow to assign work into development teams is more important than having high reusability.

*http://db.apache.org/jdo/
†https://www.jooq.org/
‡http://www.querydsl.com/
§https://github.com/enriquemolinari/inverted-layered

One of the first books I read on software design, and also one of the best, is Robert C. Martin's book, Agile Software Development[42], published in 2002. There, in addition to presenting the SOLID design principles, he presents some principles of package design (or namespace or modules as we have been defined in this book). These same principles are later re-released in the book Clean Architecture[43] by the same author, with the caveat that instead of calling them *packages*, he calls them *components*, perhaps, to align with the terminology of the C4 model by Simon Brown, who wrote a chapter in the book.

Robert C. Martin (Uncle Bob) proposes the following principles to determine how to group classes into modules:

Reuse-Release Equivalence Principle (REP): The granule of reuse is the granule of release. Which means that a set of reusable classes in a package, namespace or module for a specific purpose must also be released and versioned together.

Common Closure Principle (CCP): The classes in a package should be closed together against the same kind of changes. A change that affects a package affects all the classes in that package and no other package. This is the single responsibility principle for packages. It is closely related to the idea of vertical partition, since the partition in layers or horizontal, when there are new requirements, one has to modify classes that are in all the layers. While in an application with vertical partition, in general, the new requirements generate changes within the same module (within the same partition).

Common Reuse Principle (CRP): The classes in a package, namespace or module are reused together. If you reuse one of the classes you reuse them all. All the classes that are reused together must be in the same package, namespace or module. It is a mistake to have a single module that groups classes with functionalities that are not reused for the same purpose. Because clients who require one of those features in particular, are forced to depend on all the others as well. Instead, we should generate additional modules each with the classes being reused at the same time for the same purpose. This happened with the JDK. Before Java9, we had a single *rt.jar* module where there was everything necessary to implement any application in java. So if you made a simple "Hello World" app, we had to include the full *rt.jar*. In version 9, the JDK was modularized and partitioned according to this criteria (CRP). Now it is possible for developers to only include those modules that they really need for the application they are developing.

From the above, it is clear that there is a certain tension between these principles, mainly between CCP and CRP. There is no one-size-fits-all solution. This is where we must make our decisions and select the most appropriate criteria according to our particular situation. CCP tends to generate fewer modules than CRP, with more code and not reusable. But the changes generally impact a single module. CRP tends to generate more, small, reusable modules. Again, it depends on what type and purpose of the software we are building, the criteria to use.

If we want to favor the CRP principle when structuring an application, we will probably end up designing a layered, modular application like the one shown in Figure 3.12.

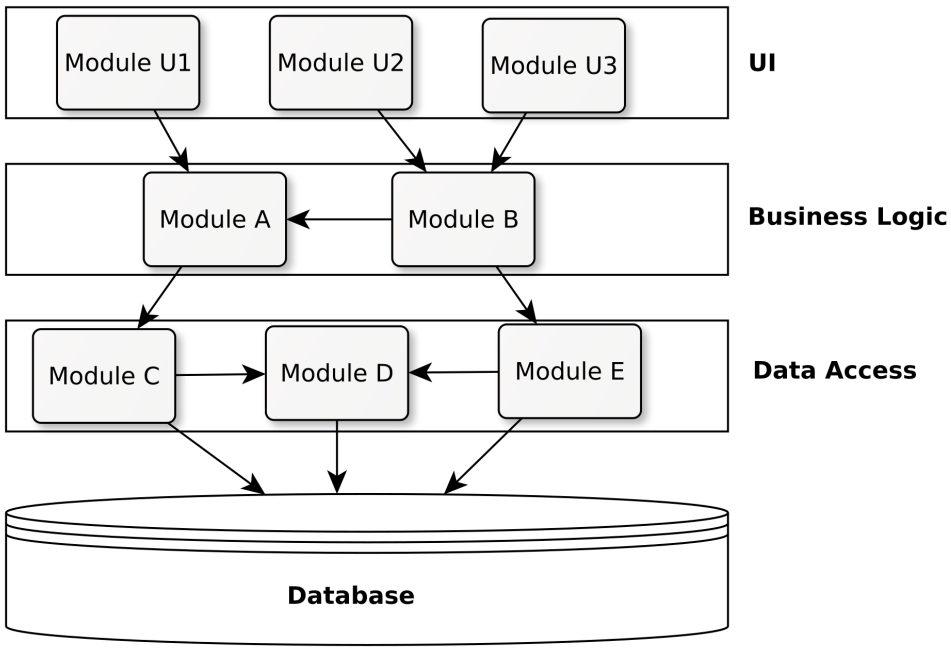

Figure 3.12: CRP Modules

If we intend to keep the complexity low of an enterprise application that we estimate is large and going to grow rapidly, we must think about modules created according to the CCP principle. Figure 3.13 shows a typical layered diagram partitioned into vertical modules, with CCP criteria.

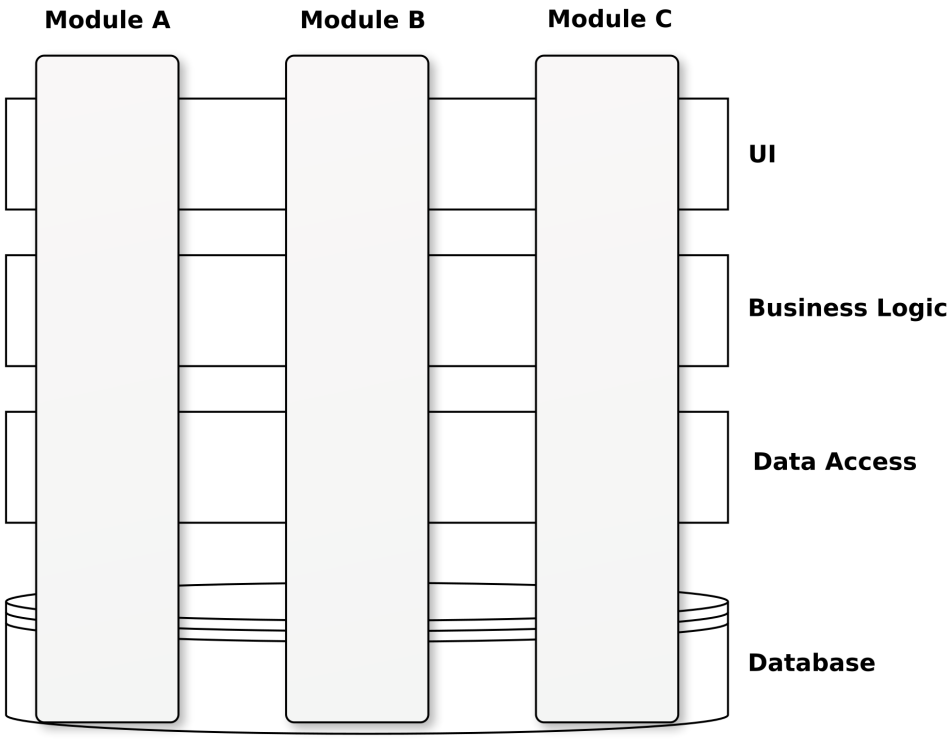

Figure 3.13: CCP Modules, Vertical Partitions

Using the CCP criteria we must identify the candidate functionalities to be implemented in different modules. Note that each vertical partition includes the persistence mechanism as well. Whether we use a relational model, or other data models, for example NoSQL, the **modules must encapsulate their persistence mechanism**. This means that the data and the decisions we make according to how to structure it are **private** to the **module**.

We might also design applications that have modules created under both principles. We could, of course, structure applications in vertical modules, but at the same time, and with the aim of not duplicating code between them, write small reusable modules (CRP) that are used by the verticals.

Implementation

With new requirements and given the current success of the radio show, we decided to implement a modular architecture with vertical partition (CCP criteria).

So let's review the requirements we have and show the new ones:

- Listeners can only register in a competition that has their registration open.
- Those participants who register on the first day of registration will receive ten points, which then can be exchanged for gifts.
- **New**: Provide a way for listeners to authenticate and register themselves in a competition.

- **New**: Every time a listener registers in a competition the system must send them an email with the confirmation of their registration.

Given the existing and new requirements we identify three modules. The **Competitions** module who will be responsible for managing competitions and registrations (including the points that the listener wins if they register on the first day of registration), the **UserProfile** module who is responsible for having the information of the listeners of the radio show and for providing an authentication service (among others). And finally the **Notifications** module responsible for sending emails when a listener registers in a competition.

We can implement the modules using a single source code repository (in a single project), where we define different source folders, one for each module or we can use a source code repository for each module (multiple projects). This discussion, which has advantages and disadvantages for both cases, is known as mono-repo and multi-repo*. We will use a source code repository for each module.

On the other hand, regardless of whether we choose mono-repo or multi-repo, we could use packages or namespaces to group classes that expose the module's public API and others packages or namespaces to group classes with the private implementation of the module. And then, use ArchUnit (or an Ant script) to validate that the dependencies between modules are exclusively through the classes defined in the public packages. But since we are using Java11, we are going to take advantage of the features that their module system provides[44] [45]. Among others, allows us to clearly define the dependencies between modules, specify what is public API, what it is not, and it is the compiler plus the execution environment that takes care that this definition is fulfilled.

Finally, we need a fourth module that we will call **Main** who will be in charge of instantiating module classes injecting their dependencies to start the application. Figure 3.14 shows the modular architecture of RadioA that we are going to implement.

*https://github.com/joelparkerhenderson/monorepo_vs_polyrepo

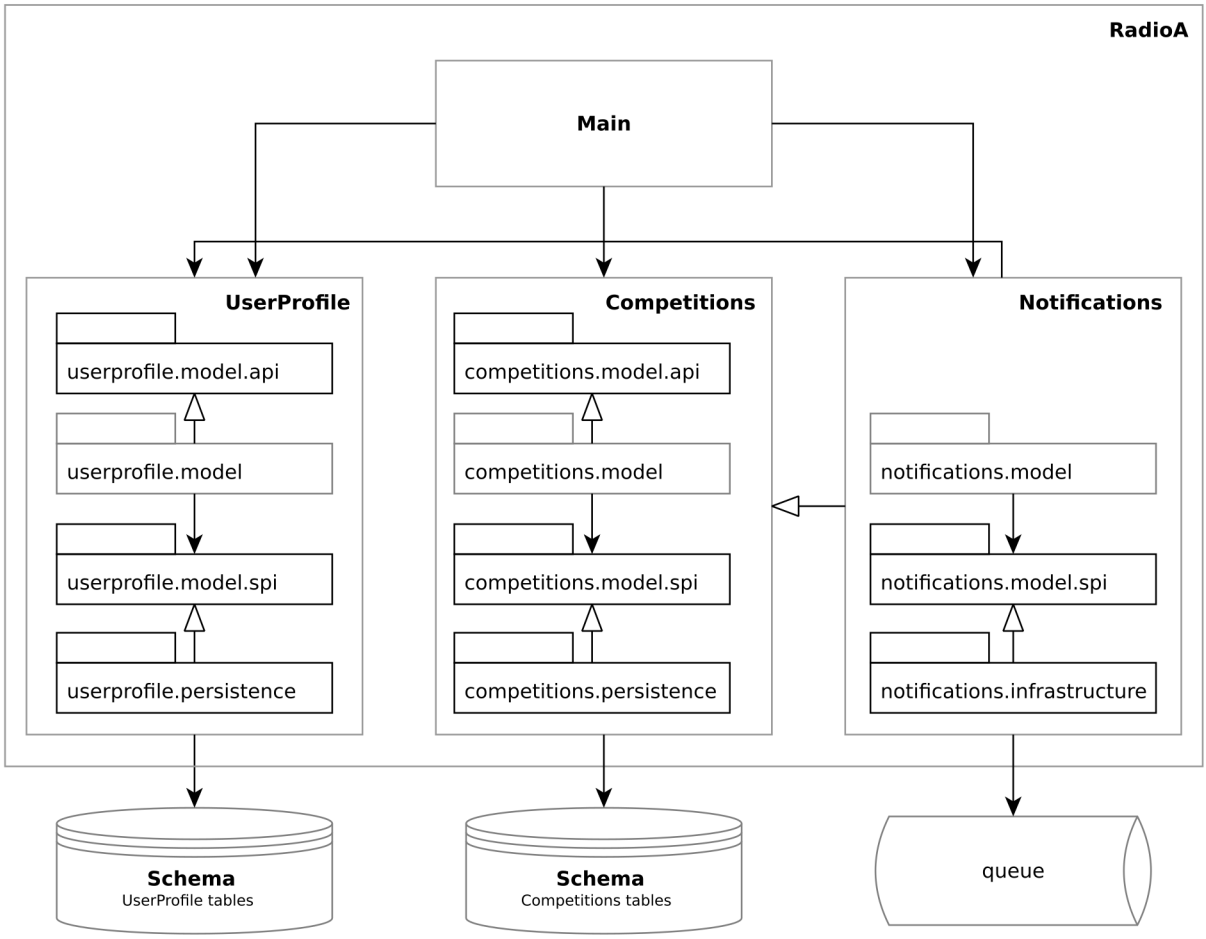

Figure 3.14: RadioA, Modular Architecture

Figure 3.14 shows a **modules/layers** diagram, as described in the previous chapter. The diagram includes the database *containers* in order to highlight that each module has its own private (inaccessible to the other modules) persistence mechanism.

Looking at figure 3.14, we can see that the **UserProfile** and the **Competitions** module does not depend on other modules. The **Notifications** module depends on the **Competitions** and **UserProfile** module as require from them services to do their job as we will see later. The **Main** module depends on the three modules of the application, as it is responsible for instantiating the application. Finally, note that each module implements the hexagonal architecture style.

Each module in the Java9+ module system must have a *descriptor* named *module-info.java*[46] which is used to define the services that the module exposes and the dependencies required to work.

Let's move to the implementation. We will describe the architectural decisions, the services offered by each module and in detail how the dependencies between modules are defined and implemented. The full sources for the four modules can be found at github/radioa-competition*, github/radioa-

*https://github.com/enriquemolinari/radioa-competition

notifications*, github/radioa-userprofile† and github/radioa-main‡.

We will start with the **UserProfile** module:

UserProfile's Module Descriptor

```
1 module radioa.userprofile {
2  exports userprofile.model.api;
3  provides model.api.RadioListeners with persistence.RadioListenersProvider;
4  provides model.api.UserAuth with persistence.UserAuthProvider;
5  //others requires
6  requires java.sql;
7   ...
8 }
```

In the **UserProfile** descriptor we can see that this module *exports* all the public interfaces of the *model.api* sub-package (I do emphasize *public* interfaces as if you have package protected interfaces they won't be exported). *model.api* is a sub-package that only contains interfaces (ports of the hexagonal architecture). This means that whoever depends on this module will only be able to access the classes (in this case only interfaces) defined there. On the other hand, the implementation of the *model.api.RadioListeners* and *model.api.UserAuth* interfaces are provided through the *RadioListenersProvider* and *UserAuthProvider* factories, using the statement *provides* (lines 3 and 4). One of the benefits offered by the Java9+ module system is that it allows clients of modules to consume services exposed by modules just referencing the public interfaces (without depending on implementation classes). This can be done for clients by using the *ServiceLoader* class (later we will show how it is used). Finally, we have the necessary dependencies for this module, defined by the *requires* statement, in this case *java.sql*, among others, to be able to use a relational database as a persistence mechanism.

Let's now go on to show the services this module offers and its data model.

UserProfile's Services

```
1  package userprofile.model.api;
2
3  public interface RadioListeners {
4   void newListener(String personId, String name, String surname,
5     String phone, String email, String userName, String pwd);
6   Optional<RadioListener> listener(int listenerId);
7  }
8
9  public interface UserAuth {
10  String authenticate(String user, String password) throws RadioException;
```

*https://github.com/enriquemolinari/radioa-notifications
†https://github.com/enriquemolinari/radioa-userprofile
‡https://github.com/enriquemolinari/radioa-main

```
11   void validate(String token) throws RadioException;
12  }
```

UserProfile's Data Model

```
1  CREATE TABLE listener (
2     id_listener INTEGER NOT NULL,
3     name VARCHAR(255),
4     surname VARCHAR(255),
5     phone VARCHAR(255),
6     email VARCHAR(255),
7     person_id VARCHAR(255),
8     PRIMARY KEY (id_listener)
9  );
10
11 CREATE TABLE listener_user (
12    id_user INTEGER NOT NULL,
13    id_listener INTEGER references listener(id_listener),
14    creation_date timestamp,
15    username VARCHAR(255),
16    password VARCHAR(255),
17    PRIMARY KEY (id_user)
18 );
19
20 CREATE TABLE token (
21    id_user INTEGER references listener_user(id_user),
22    creation_date timestamp,
23    token VARCHAR(255)
24 );
25
26 CREATE TABLE user_audit (
27    id_user INTEGER references listener_user(id_user),
28    login_date timestamp
29 );
```

Among its services it offers the creation of new listeners and getting a listener by its identifier. It also allows the listeners to authenticate in the system using username and password. If the listener is correctly authenticated, the service returns a token (with a duration of one day) to be validated with each request. Additionally, each time a listener authenticates a simple audit trail is generated.

Let's continue with the **Competitions** module:

Competition's Module Descriptor

```
module radioa.competition {
 exports competition.model.api;
 //only for injection
 exports competition.model to radioa.main;
 exports competition.persistence to radioa.main;
 //others requires
 requires java.sql;
  ...
}
```

The descriptor of the **Competitions** module exports the interfaces from the *model.api* package just like **UserProfile** does, but it also exports its private packages **only** to the **Main** module, who needs them to perform dependency injection as we will see later. In addition, it also requires *java.sql*, among others, to be able to use a relational database as a persistence mechanism.

Let's now go on to show the services it offers and its data model.

Competition's Services

```
package competition.model.api;

public interface RadioProgram {
 Iterable<RadioCompetition> availableCompetitions();
 void addInscription(int idCompetition, int idCompetitor);
}

public interface Notification {
 void send(int idListener, String message);
}
```

Competition's Data Model

```
CREATE TABLE competitor (
   id_listener INTEGER,
   points INTEGER,
   PRIMARY KEY (id_listener)
);
CREATE TABLE competition (
   id INTEGER NOT NULL,
   description VARCHAR(255),
   rules VARCHAR(1500),
   start_date timestamp,
```

```
    inscription_start_date timestamp,
    inscription_end_date timestamp,
    PRIMARY KEY (id)
);
CREATE TABLE inscription (
    id_competition INTEGER references competition(id),
    id_listener INTEGER references competitor(id_listener),
    inscription_date timestamp
);
```

Note that the **Competitions** module is responsible for the competitions and inscriptions, but the data of the listeners is kept in the **UserProfile** module. It also publishes the *model.api.Notification* interface (SPI), so that the **Notifications** module can implement it and provide the service.

Remember that one of the new requirements is that when a listener is registered in a competition an email must be sent with the confirmation. The logic for registration is implemented in the **Competitions** module, but we decided to put the implementation for sending emails in another module. In the future we could have other events where notifications of some kind are made from any other module. And it might be also possible that technical decisions about how to implement the notification will change.

At some point we had to decide whether **Notifications** should depend on **Competitions** or **Competitions** on **Notifications**. And as you saw we have decided to make the Notification module dependent on the Competition module. Whenever possible it is better to separate the abstractions from the implementations, that is, place the abstraction in the client which in this case is the **Competitions** module and implement it in other module which for us is the **Notifications** module. This gives us maximum flexibility, because it allows us to easily change the implementation of how a notification is sent without modifying or redeploying the **Competitions** module. On the other hand, the **Competitions** module implements a key business part of the application, while **Notifications** has more to do with a technical aspect of how the notification will be done. It is the same criteria used in the hexagonal architecture where technical details depend on business abstractions.

Whenever we can, the abstractions **should go in the client**, who uses the service, while their implementation should go in another module, package or namespace. These abstractions are what we call SPI (service provider interface). This is also mentioned in Kirk Knoernschild's book about modular architecture[47], as a pattern called "Separate Abstractions".

Let's see then how the SPI interface is used to send notifications:

NotificationRadioProgram Decorator

```
1  package competition.model;
2
3  public class NotificationRadioProgram implements RadioProgram {
4   private RadioProgram radioProgram;
5   private Notification notification;
6   public NotificationRadioProgram(RadioProgram radioProgram,
7     Notification notification) {
8    this.radioProgram = radioProgram;
9    this.notification = notification;
10  }
11  @Override
12  public Iterable<RadioCompetition> availableCompetitions() {
13   return radioProgram.availableCompetitions();
14  }
15  @Override
16  public void addInscription(int idCompetition, int idCompetitor) {
17   radioProgram.addInscription(idCompetition, idCompetitor);
18   notification.send(idCompetitor, "Inscription done succesfully");
19  }
20 }
```

We can see that *NotificationRadioProgram* is a decorator[48] of *DefaulRadioProgram*, which adds to the registration of a listener in a competition the statement for sending the notification. An implementation of the **competition.model.api**.*Notification* interface will be injected by the constructor of this decorator. Note that after registering the listener using the *radioProgram.addInscription*(...) statement at line 17, the notification is sent with the *notification.send*(...) statement. As it is written, it could happen for some exceptional reason that the listener is registered in the competition but sending the notification fails, since these two operations are not being executed atomically. To guarantee atomicity, we could use the outbox* pattern. In this case and for now it is not necessary, we can afford to have some notification not sent.

Let's continue with the **Notifications** module:

*https://microservices.io/patterns/data/transactional-outbox.html

Notification's Module Descriptor

```
1  module radioa.notifications {
2   exports notifications.model to radioa.main;
3   exports notifications.infrastructure to radioa.main;
4   requires radioa.userprofile;
5   requires radioa.competition;
6   //Others requires
7   requires leansoft.bigqueue;
8  }
```

This descriptor first exports its implementation to **Main** (lines 2 and 3) and then defines that depends using the *requires* statement, on the **UserProfile** and **Competition** modules, as it is also illustrated in figure 3.14. Note that the requiring UserProfile and Competitions modules means that Notifications will only have access to the module.**model**.**api** classes as this is the only exported package by these modules.

In addition, it also requires leansoft.bigqueue*, which is a message queue used to store notification messages which are dispatched in an asynchronous way.

Now we have to implement the **competition**.**model**.**api**.*Notification* SPI interface that exposes the **Competitions** module. For this we define the following class:

Notification Implementation

```
1  package notifications.model
2
3  import competition.model.api.Notification;
4  import userprofile.model.api.RadioListeners;
5
6  public class EmailNotification implements Notification {
7   private Queue queue;
8   private RadioListeners listeners;
9   public EmailNotification(Queue queue, RadioListeners listeners) {
10   this.queue = queue;
11   this.listeners = listeners;
12  }
13  @Override
14  public void send(int idListener, String message) {
15   var listener = listeners.listener(idListener);
16   var person = listener
17     .orElseThrow(() -> new RuntimeException("idListener not found..."));
18   queue
19     .push(new DefaultMessage(idListener, Map.of(DefaultMessage.EMAIL_ITEM,
```

*https://github.com/bulldog2011/bigqueue

```
20          person.email(), DefaultMessage.MSG_ITEM, message)));
21      }
22  }
```

EmailNotification is the class that provides the notification service, implementing the **Competitions**.*model.api.Notification* interface. The implementation also requires an implementation of the **UserProfile**.*model.api.RadioListeners* to be able to obtain the listener's email (which belongs to the UserProfile module) in order to generate an entry in the message queue with the message to send. Note the *import* statements (lines 3 and 4) which indicates the dependency that the **Notifications** module has with the modules **Competitions** and **UserProfile**, as figure 3.14 depict. The **Notifications** module also provides the necessary classes to consume the messages from the queue and dispatch the emails.

To finish, let's see how we inject these dependencies to be able to instantiate the application with modular architecture. To show this, lets review the module descriptor of **Main**.

Main's Module Descriptor

```
1  module radioa.main {
2   requires radioa.userprofile;
3   requires radioa.competition;
4   requires radioa.notifications;
5
6   uses userprofile.model.api.RadioListeners;
7   uses userprofile.model.api.UserAuth;
8  }
```

Note that this module requires the other three modules (lines 2, 3 and 4) and from the descriptor of the other modules we know that both **Notifications** and **Competitions** export only to **Main** their private packages, in order to be able to instantiate implementation classes to perform the injection. It also specifies the use, by means of the *uses* statement (lines 6 and 7), of the default implementations of the services provided by the **UserProfile** module.

Injecting Dependencies and Start the Application

```
1   //Obtaining RadioListeners default implementation provided by UserProfile module
2   var listeners = ServiceLoader.load(RadioListeners.class).findFirst().get();
3
4   var radioProgram =
5    new NotificationRadioProgram(
6     new DefaultRadioProgram(
7      new JdbcCompetitionRepository(user, pass, connString)
8     ),
9     new EmailNotification(
```

```
10      new BigQueueNotification("/home/enrique", "queueEmails"), listeners
11     )
12    );
13
14   //Do inscription...
15   radioProgram.addInscription(....);
```

Note how the *ServiceLoader* class mentioned above is used (line 2). It allows us to obtain the implementation of the interface **UserProfile**.*RadioListeners*, without requiring it to be coupled to it.

The **Main** module here it's just to show how dependencies are injected. In this class we would also be able to set up a web service and expose the APIs of each module as *https* services. It is also possible to use a dependency injection framework like Guice* to do the work that the **Main** module is doing here. In general these frameworks use reflection† with which in the descriptors of each module it is necessary to open the private packages by using the *opens* statement in order to make them accessible to these frameworks. This is explained in Sander Mak and Paul Bakker's book[49].

As we have mentioned, to validate this modular architecture there is no need for Ant scripts or ArchUnit tests. The Java module system does the validation for us according to the definition we write in module's descriptors. However, we must have ArchUnit tests or any other type of validation mechanism to verify package dependencies inside each module, as we showed in previous sections.

With this we finish the chapter on **module/layer** structures. We saw how to implement the layered architecture style, the hexagonal, the variant called inverted layers to finish with an implementation of a modular architecture. In the next chapter we will look at monolithic and distributed architectures, which we consider to be part of *container* structures.

Notes

29 There are some open issues to be fixed: https://github.com/TNG/ArchUnit/issues/446

30 Java 9 Modularity, Patterns and Practices for Developing Maintainable Applications. Sander Mak and Paul Bakker. 2017.

31 Patterns of Enterprise Application Architecture, Martin Fowler. 2002.

32 Fundamentals of Software Architecture: An Engineering Approach 1st Edición. Mark Richards, Neal Ford. 2020.

33 Multitier Architecture.

34 Patterns of Enterprise Application Architecture, Martin Fowler. 2002.

35 Domain Model.

36 Anemic Model.

37 Patterns of Enterprise Application Architecture, Martin Fowler. 2002.

*https://github.com/google/guice
†https://www.oracle.com/technical-resources/articles/java/javareflection.html

38 copypaste/transparent-persistence.

39 copypaste/more-on-transparent-persistence.

40 On the Criteria To Be Used in Decomposing Systems into Modules. D. L. Parnas. Carnegie-Mellon University. 1972.

41 Java 9 Modularity, Patterns and Practices for Developing Maintainable Applications. Sander Mak and Paul Bakker. 2017.

42 Agile Software Development, Principles, Patterns, and Practices. Robert C. Martin. 2002.

43 Clean Architecture, A Craftsman's Guide to Software Structure and Design. Robert C. Martin. 2017.

44 Understanding Java 9 Modules

45 Java 9 Modularity, Patterns and Practices for Developing Maintainable Applications. Sander Mak and Paul Bakker. 2017.

46 Understanding Java 9 Modules

47 Java Application Architecture: Modularity Patterns with Examples Using OSGi. Kirk Knoernschild.

48 Design Patterns: Elements of Reusable Object-Oriented Software. Gang of Four. Addison-Wesley Professional, 1994.

49 Java 9 Modularity, Patterns and Practices for Developing Maintainable Applications. Sander Mak and Paul Bakker. 2017.

4. Monolithic and Distributed Architecture

In this chapter we will look at **monolithic** and **distributed** structures and their styles.

Monolithic Architectures

Definitions

First I would like to clarify that a **monolithic architecture does not mean spaghetti code**. With the growth of Microservice architecture it seems (and also helped by some authors) that monolithic architectures are now an anti-pattern or a bad practice. The only bad practice in all of this is spaghetti code. Every architectural decision has advantages and disadvantages. "Everything in software architecture is a trade-off"[50].

We call monolithic architecture (a term that takes prominence with the arrival of Microservices) to an application that is deployed in a single deploy unit, running in a single process where its elements communicate in the form of in-process. While in distributed architectures the application is deployed with several deploy units, each one running in its own process and they communicate with each other in the form of inter-process (IPC).

Modular Monolith

Let's continue with the RadioA application that we designed with modular architecture in the previous chapter and whose module diagram is shown in figure 3.14. Suppose then that we modify the **Main** module to expose the services of the **Competitions** and **UserProfile** modules as *https* services using a lightweight framework like Javalin*. Then suppose we implement a single-page Web front-end and a mobile application where both consume the *https* services that **Main** exposes.

Lets illustrate this with the *containers* diagram of figure 4.1:

*https://javalin.io/

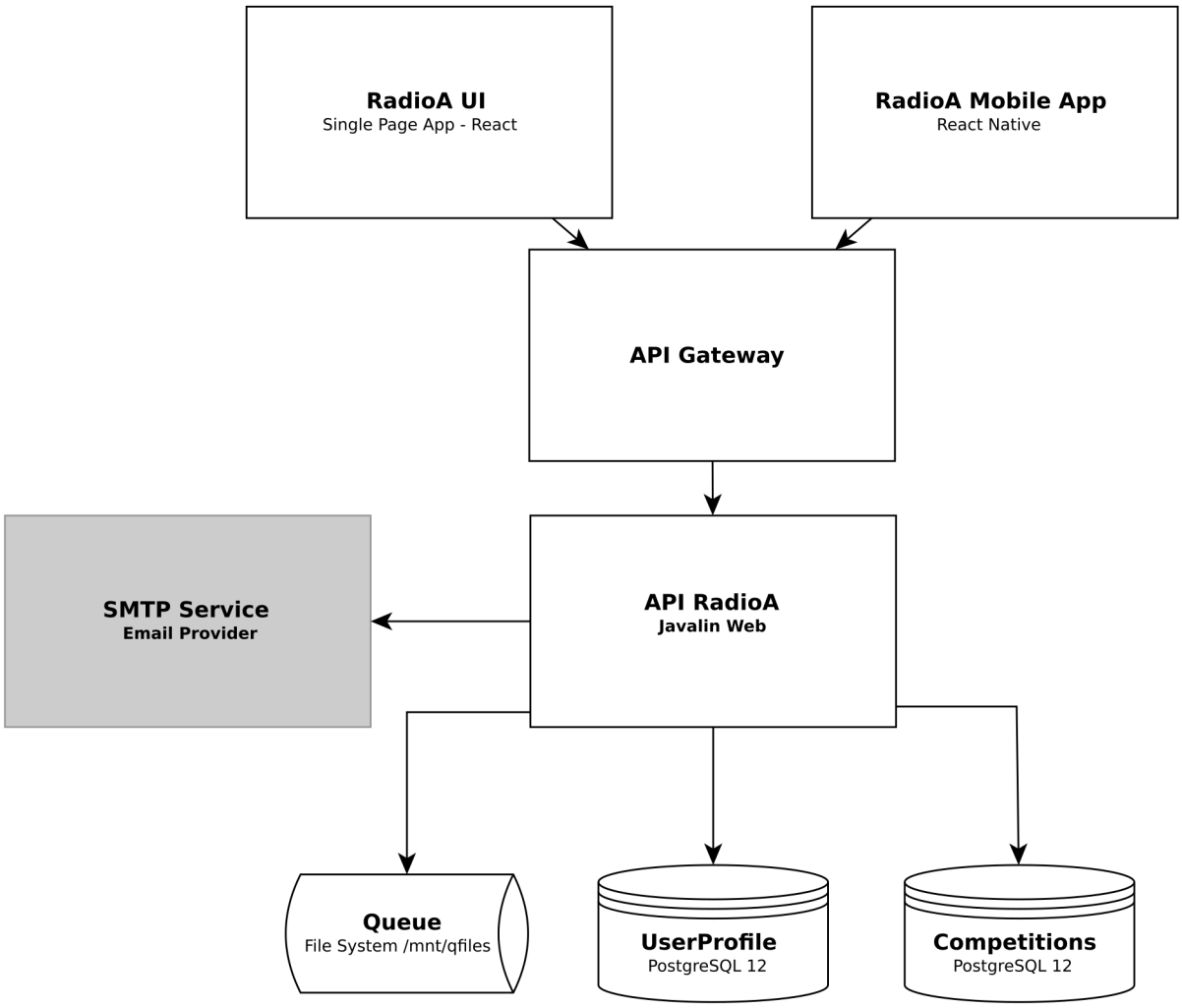

Figure 4.1: Modular Monolith

The **API RadioA** (Javalin Web) box refers to the RadioA application (whose modular architecture is illustrated in figure 3.14) at run-time, deployed in a single artifact. The diagram shows also the databases and the message queue containers that each module requires as explained in the previous chapter. In addition, you can see others containers, the mobile front-end and the Web single-page front-end, communicating to the **https** services through an API Gateway*.

As illustrated by the figure 4.1, our RadioA application with modular architecture deployed as a single artifact is what is known as **Modular Monolith**. The term Modular Monolith was popularized by Simon Brown[51] the author of the C4 model that we described in chapter 2. This term was mainly thought of with the idea of differentiating an application with good architecture (modular), from a monolithic application with spaghetti code.

All the modules are deployed in a single container (web server in this case) and the communication

*https://microservices.io/patterns/apigateway.html

between them is in the form of in-process, as we have mentioned. Let us then go on to study distributed architectures and then make a comparison between the two.

Distributed Architectures

This is perhaps one of the hottest topics in recent years, not only due to the rise of Microservices and NoSQL databases, but also due to in every organization there is always some kind of integration between different systems. We consume APIs from internal and external systems, which leads us to have to deal with the complexities of distributed systems. In this section we will begin by explaining the difficulties we encounter in distributed systems, to finally explain the Microservice architecture by implementing RadioA with that structure.

Reasons to Distribute

In distributed architectures, a system is divided into multiple smaller applications where each one is deployed in its own container, runs in its own process and communicates with each other by sending messages through the network as shown in the figure 4.2.

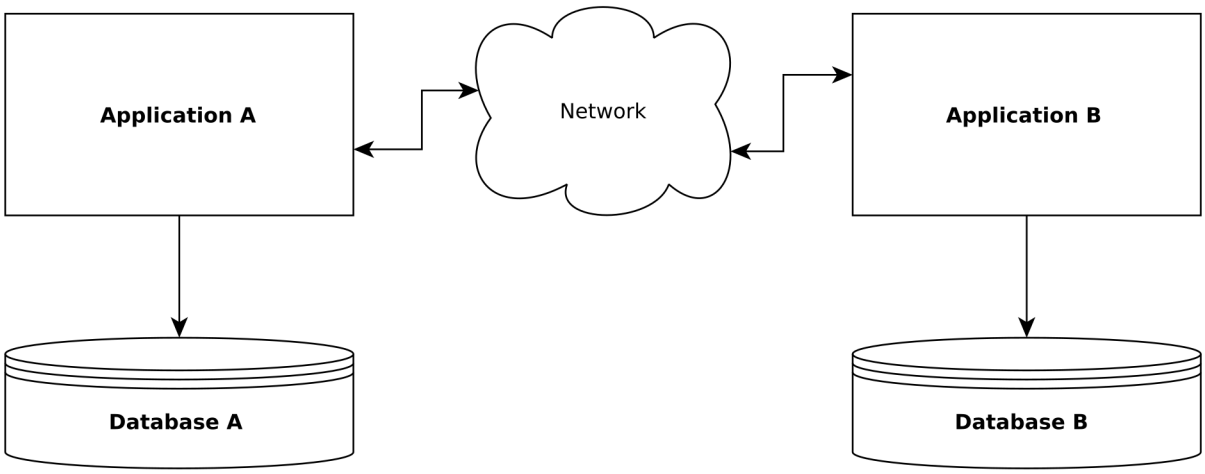

Figure 4.2: Distributed Architecture - Communication

As you may have noticed in figure 4.1, between the browser, or a mobile application and the back-end application (or server side) we already have communications through the network between distributed applications. But here we go deeper, we split and distribute the back-end (represented by the **API RadioA** box), which complicates things even more.

So, if it complicates everything even more, why do we distribute a system?

As we mentioned, everything in architecture is a trade-off. While distributing a system complicates things on one side it will give us benefits on other sides. In this case, by distributing we improve the

performance, **scalability** and **availability**. It is not always possible to upgrade a server's hardware to improve **performance**, and at some point as well, the cost of upgrading it is prohibitive. By distributing a system, performance can be improved since the load is distributed over two or more commodity servers. If the system we build is successful, the number of clients will grow, forcing us to handle more requests and more volume of data. By distributing the load and the volume of data on different computers, allows us to **scale** to levels impossible to achieve in monolithic architectures. The only way to improve the **availability** of our system is by replicating it on multiple computers so that if one fails, another can replace it. To do that we need to distribute the system. We will cover these quality attributes in details in the following chapters.

The mentioned advantages come with a set of difficulties and limitations that we do not find in monolithic architectures. It is important to understand what they are in order to make the right decisions to handle them.

Partial Failures

When an application is running on a single machine, its behavior is predictable. If the hardware where it runs works, the application will work. If the hardware fail, for any reason, the entire application will stop working. That is, an application on a single machine works or does not work, but there is nothing in between (removing from the equation possible bugs that the application could have).

When we design an application to run on multiple machines communicating over a network, the situation is very different. Distributed applications are made up of several elements that could fail. The network, along with all its complexity, plus the machines where the application runs are part of the system. Any of these elements (computers, network devices such as routers, switches, etc.) that are part of a distributed system might fail while other elements of the system might be working fine. This is known as **partial failure**[52].

The problem is that these partial failures can occur in any of the elements of the system in an unpredictable way and behave differently in each case. Partial failures are the main difficulty we face in distributed architectures. Of course, these failures could be very rare. But if they do occur we have to be able to handle them, designing the applications with this in mind.

In general, these difficulties are ignored by those developers who are new to the implementation of distributed systems. Because of this, a technical team working at Sun Microsystems in the 1990s published a set of fallacies made by inexperienced developers when implementing distributed systems. These are known as the fallacies* of distributed computing and are intended to help better understand the difficulties caused by partial failures.

The fallacies are listed below:

1. The network is reliable.
2. Latency is zero.

*https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

3. Bandwidth is infinite.
4. The network is secure.
5. Topology does not change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

From these fallacies, we are more interested in understanding better why the **network is not reliable**. Networks do not have the ability to guarantee when a packet will reach its destination and cannot even guarantee that it actually reaches its destination. The book Designing Data-Intensive Applications[53] is very clear describing the possible situations that could happen when an application A sends a message to an application B over the network:

1. The request might have been lost (perhaps someone unplugged a network cable).
2. Your request may be waiting in a queue and will be delivered later (perhaps the network or the recipient is overloaded).
3. The remote node may have failed (perhaps it crashed or it was powered down).
4. The remote node may have temporarily stopped responding (perhaps it is experiencing a long garbage collection pause), but it will start responding again later. See figure 4.4.
5. The remote node may have processed your request, but the response has been lost on the network (perhaps a network switch has been misconfigured). See figure 4.3.
6. The remote node may have processed your request, but the response has been delayed and will be delivered later (perhaps the network or your own machine is overloaded).

The usual way to deal with this issue is by using **timeouts**. That is, after a certain time it is assumed that no response will be received. However, this does not guarantee that the remote machine has received the message or not. In Figure 4.3, application A sends a message to application B, which receives the message and processes it. But the response is lost in the network, it does not reach application A. After a certain time (according to the configured timeout), application A terminates the communication as failed. In figure 4.4 something similar happens, the message reaches application B, but for some reason it takes a long time to process it, exceeding the timeout configured in application A. With which, as in the previous case, application A terminates the communication as failed, ignoring the answer that eventually comes later. In both cases, application A terminates with an error, without being able to differentiate between the loss of the response in the network (figure 4.3) or the delay in processing (figure 4.4).

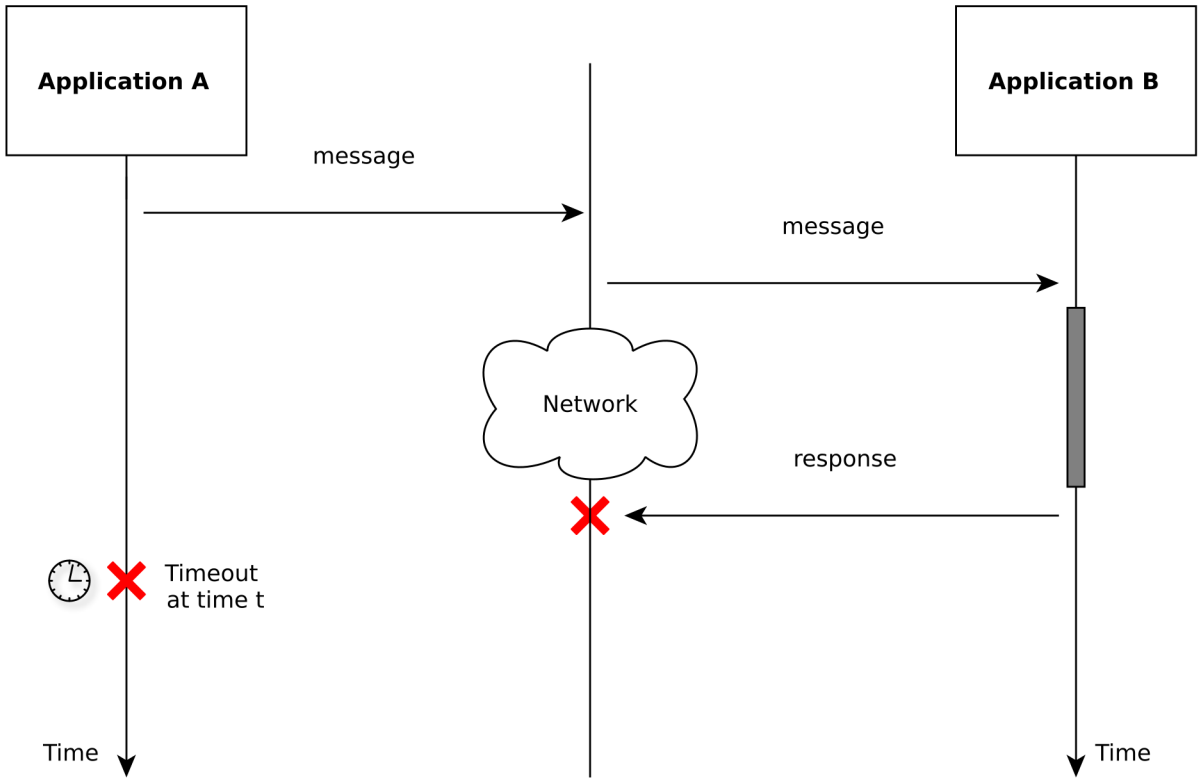

Figure 4.3: response is lost in the network

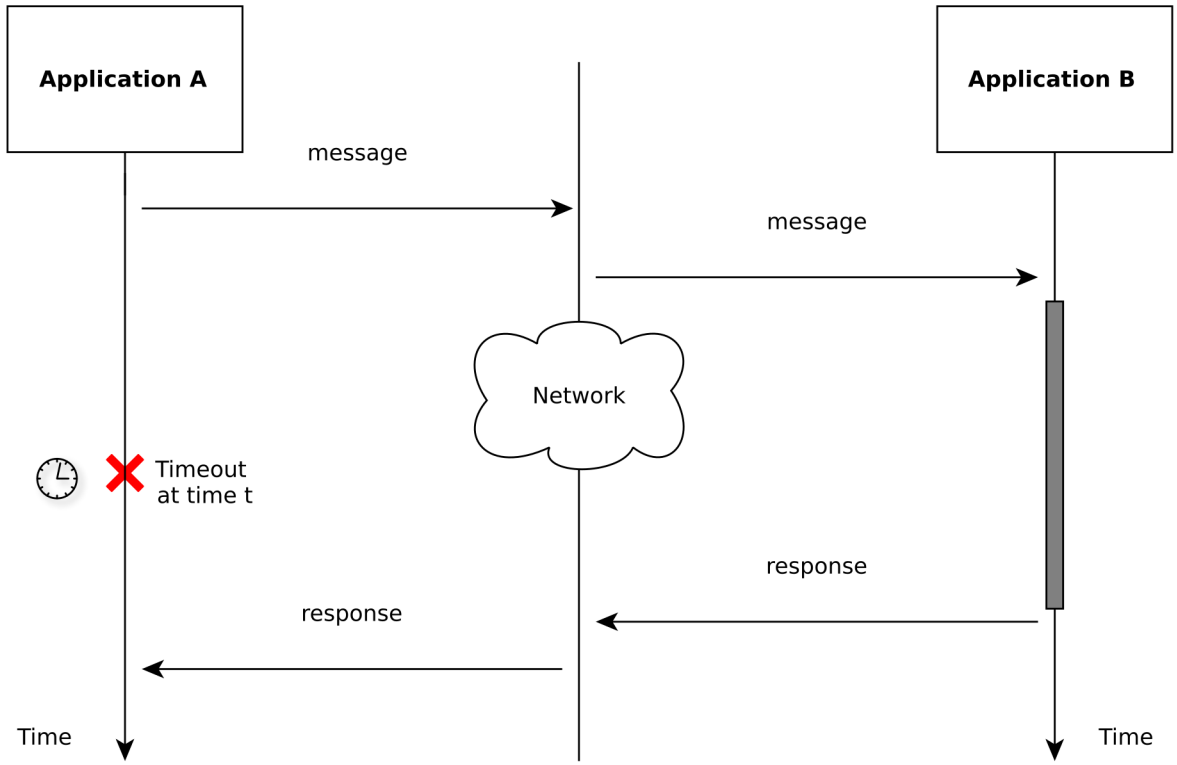

Figure 4.4: Too much time processing

In the scenarios illustrated in figure 4.3 and 4.4 the application A will try to send the message again, due to the reception was not confirmed. This situation is described as **exactly-once semantics**. Here it is important to understand what effects produce on the system the fact that a message might be sent more than once. Perhaps the message is simply to send an email and if it is sent twice with the same content is not important. Or perhaps the message is to make a debit from a bank account with which we need to be sure that it will not happen more than once by mistake.

Understanding the consequences of **partial failures** allows us to design applications knowing that the same message might be sent more than once. One way to deal with this issue is to make services **idempotent**, that is, if they are executed more than once with the same inputs, they do not produce any other effect than the one produced the first time. The other way to handle it is known as **de-duplication**, where each message is given a unique identifier. With each forwarding of the same message the same identifier is used. In this way, the receiver could check if the message was already received to discard it if that was the case.

In a **synchronous** communication model, like the one illustrated in the previous figures, application A, after sending the message, waits for a response. It only continues the execution if it receives a response from application B or the configured timeout elapses.

If we wanted to prevent application A from waiting for a response from application B, we should

perform the communication **asynchronously**. For example, the message might be persisted in a queue where then another process will take care of it and send it to application B. Here we also need to deal with timeouts and face the problems caused by the possibility of sending the same message more than once, but the advantage is that application A could continue executing and attending other requests.

Transactions

In centralized systems, thanks to the **transaction** concept implemented by relational databases (and some NoSQL) we have the property called **atomicity** guaranteed. In other words, if there are two or more operations to be carried out, both will be executed or neither of them will do in the event that one of them fails. Databases that implement this concept free application developers from having to deal with this, which implies simpler software systems. In distributed systems, guaranteeing atomicity is somewhat more complex. We call **distributed transactions**, when we involve two or more computers in operations that must be executed atomically. We could say that in distributed architectures we have two variants[54] where we need to guarantee atomicity, one more complex than the other.

The first occurs when we need certain information stored in a database to be replicated to other databases. Relational databases have built-in mechanisms to carry out this type of work (master/slave replication) or in case we are talking about databases with different data models because they belong to different applications, we could use publish/subscribe mechanisms to do replications. In general, data replication is done asynchronously, so as not to delay the end users while the information is copied. This produces what is called **eventual consistency**. That is, at some point all the databases will be consistent.

The second variant, and more complex, occurs when we need to update different information that resides in different databases atomically. For example, suppose a Web application that takes hotel room reservations. To confirm the reservation, first we must validate if the customer's credit card is valid, then generate the invoice and send it by email. These three actions occur in three different applications and must be performed atomically. To achieve this there is a technique called **Sagas**[55]. Basically, the *saga* consists of a sequence of transactions T_1, T_2, ..., T_n, where each transaction T_i is associated to a **compensating** transaction C_i which is executed in case a rollback is required. Figure 4.5 illustrates this concept.

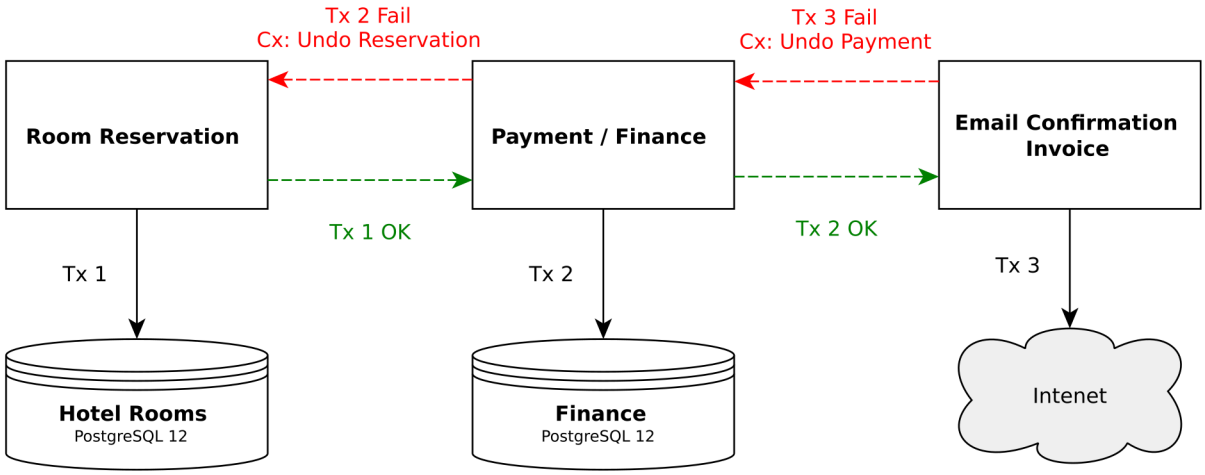

Figure 4.5: Sagas

As the figure 4.5 shows on each successful Tx_i transaction, a message is sent to the next application involved in the *saga*. If any of these transactions T fail, the compensating transactions are started in order to undo what was done up to the point of failure. In general, messages involved in the transactions (Tx_i or Cx_i) are implemented asynchronously via a publish/subscribe mechanism.

There are other mechanisms[56] to perform distributed transactions, however, *sagas* is the mechanism recommended for Microservices architectures.

Distributed Monolith

Before going to Microservices, it is important to understand the architecture style known as **Distributed Monolith**. This style suggests splitting an application in multiple simpler applications where each one runs in its own process, they communicate with each other in the form of inter-process (IPC) and **synchronously**.

Suppose we take the RadioA application with modular architecture and we convert each of the modules into a simple Web application that exposes *https* services. That is, each service that modules expose and can be invoked by method calls we now expose them via *https* using Javalin*. We also need the **Notifications** module to expose the notification service via *https* as well. Then each of these modules is deployed in its own *container* communicating with each other via *https* synchronous calls. Let's see then the container diagram of RadioA with distributed monolith architecture:

*https://javalin.io/

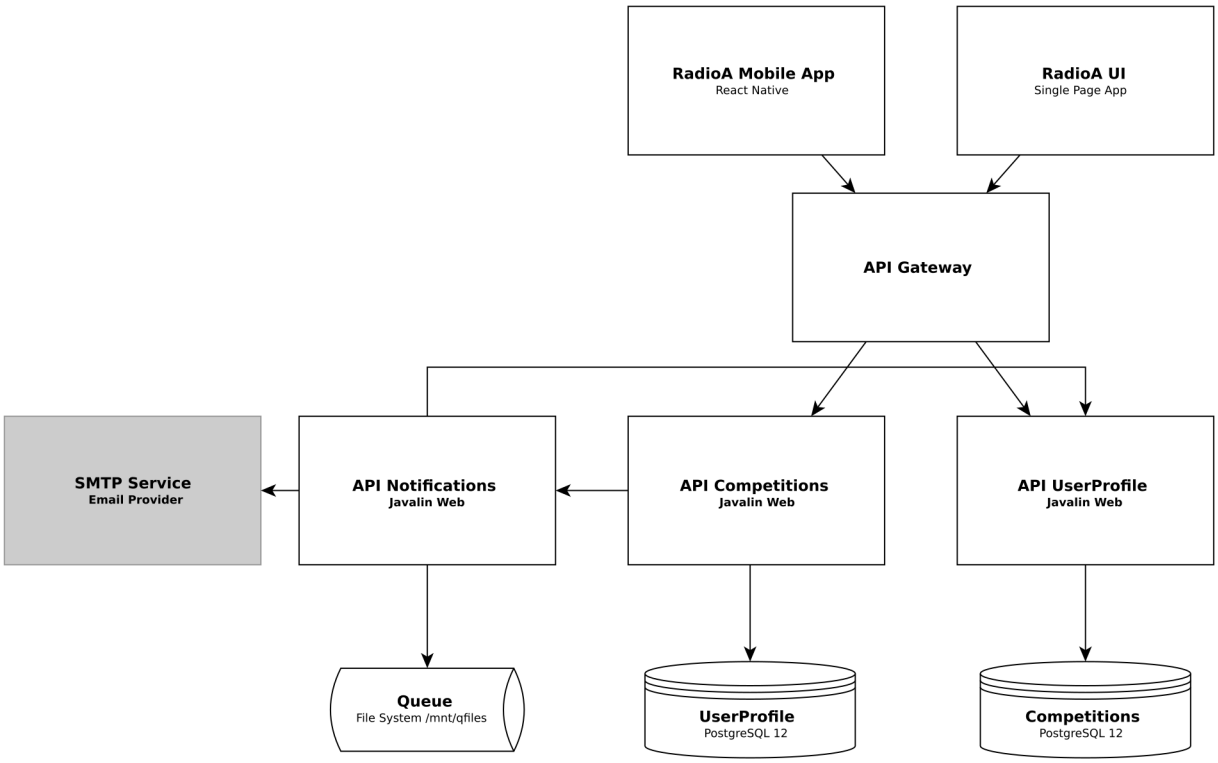

Figure 4.6: Distributed Monolith

If we look at figure 4.6 we can see the two front-end containers, the single-page application and the mobile application, have not changed with respect to figure 4.1. Then we can see the three modules converted into Web applications: API **Notifications**, API **Competitions** and API **UserProfile**. All of them exposing *https* services. Note the difference with figure 4.1, where API **RadioA** was the only monolithic deployment unit and now we have three. The arrows between the different containers in this case are all *https* synchronous communications. **Competitions** consume a service from the **Notifications** application to send an email after an inscription is done. And **Notifications** consumes a service from **UserProfile** to retrieve the email from the listener required to send the notification.

This architecture style is considered an anti-pattern because it has all the disadvantages that distributed systems present and also **temporal coupling**. That is, to be able to register a listener in a competition, I not only need to have the **Competitions** application available but also the **Notifications** application (and this in turn the **UserProfile**). If any of them were not available, it would not be possible to register a listener in a competition. Thus, it has the cons of the modular monolith, which is the temporal coupling between modules, and the cons of the distributed systems which are dealing with partial failures. An advantage, not minor, over the modular monolith architecture is that it allows having different technologies for each application, at the entire stack. So you might have coded the UserProfile application in Ruby with a NoSQL Database and the Competitions application in PHP with MySQL.

Microservices

Definitions and Style

The Microservices architecture style, like the previous style, has similarities with modular architecture, in terms of the criteria used to split an application into simpler applications or Microservices. The difference is that we are talking about applications, not modules, and the communication between them is in the form of inter-process. So far from my description there is no difference with the distributed monolith style. So? What do Microservices Architecture have special? How does it differ from Distributed Monolith?

It proposes to remove the temporal coupling between applications, between Microservices. This style screams that Microservices should be autonomous. Microservices must be developed, tested and deployed isolated from the rest of the Microservices that form the whole system. Which as we said, in the Distributed Monolith none of these can be done.

How can this be achieved? How can I remove the temporal coupling?

We have to change the way Microservices communicates. The temporal coupling is generated by the synchronous communication. And to change that, as we do in software design, we have to put another element between the two.

We can do this by using a publish/subscribe* mechanisms for their communication, which is an asynchronous talk between them. Its main characteristic is that neither end of the communication has knowledge of the other. The one who consumes messages subscribes to a category of messages (or specific topics) and the one who produces messages, has no knowledge of who may be interested in them. This mechanism removes the temporal coupling between the Microservices.

Any form of coupling between Microservices must be removed from the system, and with that we are eliminating any possibility of code reuse between Microservices. The style by definition clearly expresses that there should be no dependencies between development teams responsible for different Microservices, and since code reuse generates dependencies between them, it is not something that is promoted. In other words, in this style of architecture, independence between teams is favored over code reuse. Sam Newman in his book Building Microservices[57] states:

"The evils of too much coupling between services are far worse than the problems caused by code duplication".

How do we split an application into Microservices then?

It is the most complex part, but guess what? We have already discussed this in the modular architecture style as the criteria for splitting an application into modules is the same used to split an application into Microservices. Indeed, as we see later, having a modular architecture gives us the possibility, if it's really needed, to move to a Microservices without much trouble.

So, the criteria is the one mentioned in the previous chapter, using the CCP (Common Closure Principle). Everything that changes for the same reason and at the same time when there is a new

*https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

requirement, must be in the same Microservice. The **goal** is *very clear*, we need that most of the requirements fall on a single team and therefore on a single Microservice. The less need to involve multiple teams to implement a requirement the faster we will be. This is why the CCP principle applies perfectly.

However, the general Microservices literature recommends using the concept called **Bounded Context**, that belongs to the Domain Driven Design[58], to determine the granularity of each Microservice. Bounded Context is designed as a solution to understand and divide an enterprise application into sub-domains according to the different responsibilities within each domain of an organization. Both bounded context and CCP generate vertical partitions, promoting the same goal that we described in the previous paragraph. It is not the intention of this book to go into detail about DDD or bounded context, for this we recommend the book by Eric Evans[59]. Also, books on Microservices deal with this topic, such is the case of Sam Newman's book[60].

In the same direction, the book Fundamentals of Software Architecture[61] proposes three elements to help architects split an application in Microservices:

Purpose: Ideally, each Microservice should be extremely functionally cohesive, contributing one significant behavior on behalf of the overall application.

Transactions: Bounded contexts are business workflows, and often the entities that need to cooperate in a transaction show architects a good service boundary. Because transactions cause issues in distributed architectures, if architects can design their system to avoid them, they generate better designs.

Choreography: If an architect builds a set of services that offer excellent domain isolation yet require extensive communication to function, the architect may consider bundling these services back into a larger service to avoid the communication overhead.

The term micro-service seems to reveal that each application should do the bare minimum. But the criteria is by sub-domain, and perhaps certain sub-domains are not small. For this reason, what is explained in the previous chapters is so important. It is necessary to have a solid architecture of modules or layers, within each Microservice. Otherwise, instead of spaghetti code in a monolithic application, we will have spaghetti code in multiple distributed applications. It is true that if the size of each application is relatively small, it is simpler to tolerate the spaghetti code.

When you refer to small or medium size in software, you are really saying that it is possible to understand where the changes that are made in each iteration will impact. So if I have an application divided into Microservices, where each Microservice lacks of a solid structure, the complexity of each application should grow more slowly than if I have a monolithic application with spaghetti code. Sooner or later, software entropy will win if we don't have a solid architecture with clear definitions (and validated at every commit).

Because the communication between the client and a Microservice is only in the form of inter-process, some books dedicated exclusively to Microservices mention this as a main advantage over monolithic architectures, saying that in the latter, a distracted developer could go from the client to the database, without going through an intermediate layer or an API. With what we have seen in the previous chapters, you might have noticed that this is far from being the case. Using the

Java9+ modules or even validating the architecture with tools like ArchUnit, one can ensure that the accesses are always as defined. Thus, this is not a reason to use this style of architecture. The reasons, as we've mentioned, are scalability, performance, and availability. When we talk about scalability it is with a dual purpose. On one hand, the characteristic that applications can support a greater demand without too much effort. And on the other hand, the ability to have development speed by eliminating the bottleneck that is generated with a single monolithic application in organizations with hundreds of developers.

Implementation

So let's see how we transform the RadioA application with modular architecture into a Microservices architecture. The vertical module partitioning that we have implemented in the previous chapter to show modular architecture is the **first step** towards a Microservices architecture. We are going to take these projects as a basis to transform each module into a Microservice.

We will have three Microservices: **UserProfile**, **Competitions** and **Notifications**. The first thing we will do is incorporate a Web *package* to the UserProfile and Competitions modules in order to expose their services via *https*. And then we will incorporate a publish-subscribe messaging mechanism to perform the communication between Microservices. Let's look at figure 4.7 below:

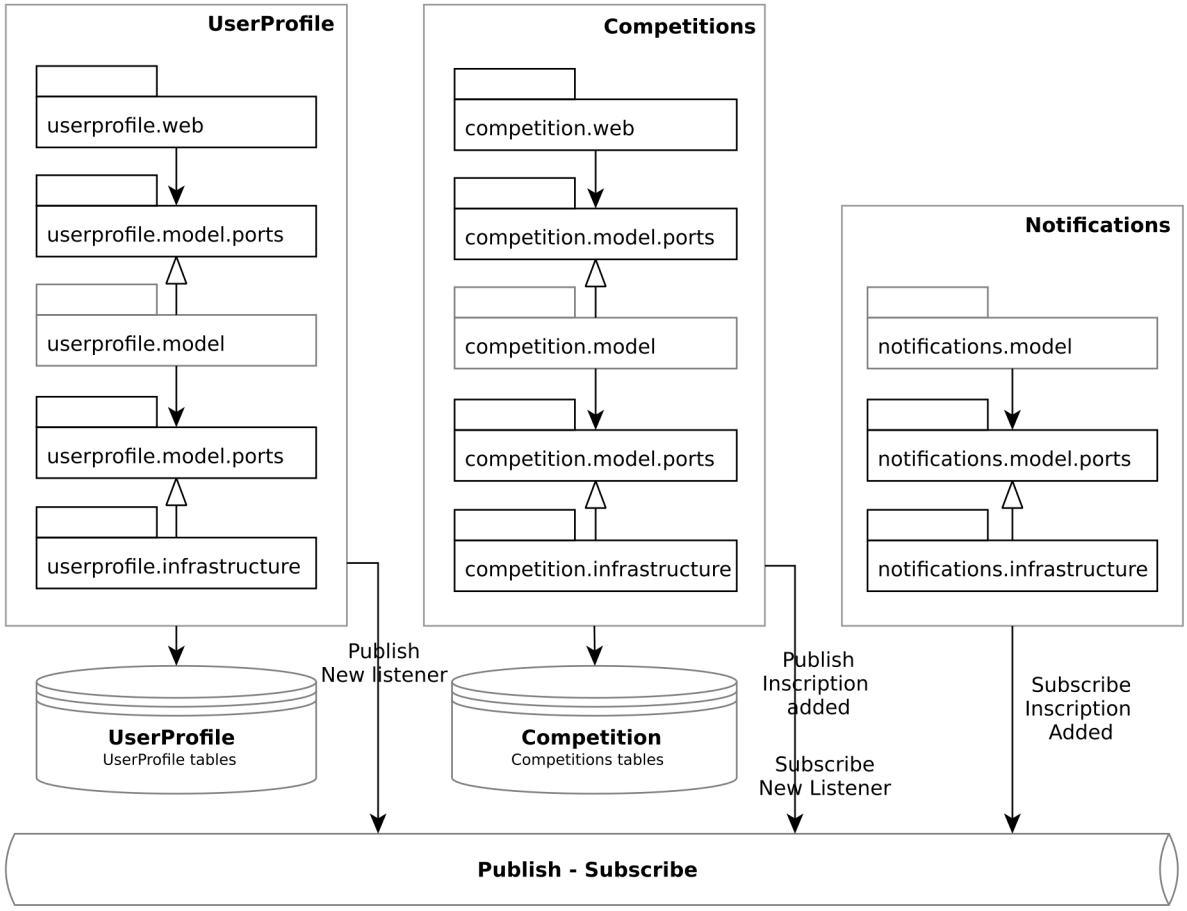

Figure 4.7: Microservices with Modules/Layers

Lets see figure 4.7. Each Microservice is implemented with the hexagonal architecture. The Microservices **Competitions** and **UserProfile** have a **web** package which exposes their services via *https*. Whereas **Notifications** is a standalone application. Let us remember that one of the requirements we had was to notify the listener via email when they register in a competition. To allow Microservices to communicate with each other, we incorporate a publish/subscribe mechanism. Who finally sends the email to the listener is the **Notifications** Microservice, but for that, it needs to know when to do it and requires the listener email address, a title and a text body to send. The Microservice that knows when someone registers in a competition is **Competitions** and to let other Microservices know that an inscription has been done, it publishes the "inscription-added" event. But who has the email address of each listener is the **UserProfile** Microservice. To share that information with whoever needs it, every time a new listener is added, the "new-listener" event is published (if we add the possibility to modify its email to the listener, a new event should be published). **Competitions** subscribes to the "new-listener" event and store the listeners' email in its relational data model. In a similar way, the action of making a new registration in a competition generates the "inscription-added" event, which includes as the event's data the email address of the listener who registered, a title and the text body. Finally, the **Notifications** Microservice subscribes to the "inscription-added"

event, sending the email to the listener each time that event arrives.

Let's go on to show part of the sources of each of the Microservices, to understand mainly how they expose their services and how they publish and subscribe to events. Let's start with the **UserProfile** Microservice.

UserProfile - Add New Listener Service

```
POST https://[host]/listener
{
 "personId": "string",
 "name": "string",
 "lastName": "string",
 "email": "string",
 "phone": "string",
 "user": "string",
 "pwd": "string"
}

Successful response:
{
 "result": "success"
}

Error response:
{
 "result": "error",
 "message": "string"
}
```

UserProfile - Retrieve a Listener by Id

```
GET https://[host]/listener/[id]

Successful response:
{
 "result": "success",
 "email": "string",
 "personId": "string",
 "lastName": "string",
 "name": "string"
}

```

```
Error response:
{
 "result": "error",
 "message": "string"
}
```

UserProfile - Authentication

```
POST https://[host]/authenticate
{
 "user": "string",
 "pwd": "string"
}

Successful response:
{
 "result": "success",
 "token": "string"
}

Error response:
{
 "result": "error",
 "message": "error message"
}
```

UserProfile exposes three services. One that adds a new listener to the database and ends up publishing the "new-listener" event. Another that retrieves a listener by its identifier and finally a service to authenticate a listener.

All the applications that we implement as a Microservice have a class *main.Main* that instantiate, injects the dependencies and starts the *https* service. Let's see how the **UserProfile** Microservice solves it:

UserProfile Main

```
package main;

public class Main {
 public static void main(String[] args) {
  String connString = Objects.requireNonNull(
    System.getProperty("conn-string"),
    "specify conn-string as a jvm argument");
  String user = Objects.requireNonNull(System.getProperty("user"),
    "specify a database user as a jvm argument");
  String pass = Objects.requireNonNull(System.getProperty("pwd"),
    "specify a database pwd as a jvm argument");
  String pubsub = Objects
    .requireNonNullElse(System.getProperty("pub-sub-server"), "localhost");
  String pubsubPort = Objects
    .requireNonNullElse(System.getProperty("pub-sub-port"), "6379");

  var tx = new Tx(connString, user, pass);

  new Web(
    new PublishNewListener(
      new DefaultRadioListeners(
        new JooqRadioListenerRepository(tx),
        new JooqUsersNames(tx)
      ),
      new RedisEvent(pubsub, Integer.valueOf(pubsubPort)))
    ),
    new DefaultUserAuth(
      new JooqTokens(tx),
      new JooqUsers(tx)
    ),
    7001
  ).start();
 }
}
```

userprofile.web.Web is the class that starts the *https* service on port 7001 and delegates to the services (ports of the hexagonal architecture from the *userprofile.model.ports* package) implemented by the application (*userprofile.model* package). Let's review the implementation of the *userprofile.Web* class:

UserProfile Web class

```
package userprofile.web;

public class Web {
 private RadioListeners listeners;
 private UserAuth auth;
 private int webPort;

 public Web(RadioListeners listeners, UserAuth auth, int webPort) {
  this.listeners = listeners;
  this.auth = auth;
  this.webPort = webPort;
 }

 public void start() {
  Javalin app = Javalin.create().start(this.webPort);
  app.get("/listener/:id", listener());
  app.post("/listener", addListener());
  app.post("/authenticate", auth());

  app.exception(RadioException.class, (e, ctx) -> {
   ctx.json(Map.of("result", "error", "message", e.getMessage()));
   // log error in a stream...
  });

  app.exception(Exception.class, (e, ctx) -> {
   ctx.json(
     Map.of("result", "error", "message", "Ups, something went wrong..."));
   System.out.println(e.getMessage());
   // log error in a stream...
  });
 }

 private Handler addListener() {
  return ctx -> {
   ListenerData dto = ctx.bodyAsClass(ListenerData.class);
   this.listeners.newListener(dto.getPersonId(), dto.getName(),
     dto.getLastName(), dto.getPhone(), dto.getEmail(), dto.getUser(),
     dto.getPwd());
   ctx.json(Map.of("result", "success"));
  };
 }
 private Handler listener() {
```

```
43    return ctx -> {
44     Optional<RadioListener> l = this.listeners
45       .listener(ctx.pathParam("id", Integer.class).get());
46     ctx.json(l
47       .map((l1) -> Map.of("result", "success", "personId", l1.personId(),
48         "name", l1.name(), "lastName", l1.name(), "email", l1.email()))
49       .orElse(Map.of()));
50    };
51   }
52   private Handler auth() {
53    return ctx -> {
54     AuthData dto = ctx.bodyAsClass(AuthData.class);
55     String token = this.auth.authenticate(dto.user(), dto.pwd());
56     ctx.json(Map.of("result", "success", "token", token));
57    };
58   }
59  }
```

The *Web.start*() method declares the *URI* of each *https* service and which private method to delegate when invoked. Note that the implementation in the private methods simply delegates to the ports that the package *userprofile.model.ports* exposes and then transforms the output into a *java.util.Map* to finally serialize it into a *json*.

The first parameter of the *Web* class constructor is *RadioListeners*. If we look again at the *Main* class, we will see that we are passing an instance of the *PublishNewListener* class (line 20 of the UserProfile Main class) that is a decorator of *DefaultRadioListener*, which publish the message "new-listener" in the publish-subscribe mechanism after storing the new listener in the database. Let's see:

UserProfile Publish

```
1  package userprofile.model;
2
3  public class PublishNewListener implements RadioListeners {
4   private static final String EVENT_NAME = "new-listener";
5   private RadioListeners listeners;
6   private Event event;
7
8   public PublishNewListener(RadioListeners listeners, Event event) {
9    this.listeners = listeners;
10   this.event = event;
11  }
12
13  @Override
14  public int newListener(String personId, String name, String surname,
```

```
15      String phone, String email, String username, String pwd) {
16    int idListener = this.listeners.newListener(personId, name, surname,
17      phone, email, username, pwd);
18    event.publish(EVENT_NAME, new NewListener(idListener, email).toJson());
19    return idListener;
20   }
21   @Override
22   public Optional<RadioListener> listener(int listenerId) {
23    return this.listeners.listener(listenerId);
24   }
25  }
```

In the *newListener* method (line 14) after storing the new listener in the database (line 16), we have the publishing of the "new-listener" event (line 18). Let's see then the *NewListener* Event class.

NewListener Event

```
1  package userprofile.model;
2
3  public class NewListener {
4
5   private static final String EMAIL = "email";
6   private static final String ID_LISTENER = "idListener";
7   private int idListener;
8   private String email;
9
10  public NewListener(int idListener, String email) {
11   this.idListener = idListener;
12   this.email = email;
13  }
14
15  public String toJson() {
16   return new Gson()
17     .toJson(Map.of(ID_LISTENER, idListener, EMAIL, this.email));
18  }
19 }
```

Note that the *NewListener* class encapsulates the identifier of the listener and her email. It provides the method *toJson*() that creates a *json* with the following structure, which is what is persisted in the publish/subscribe mechanism:

new-listener Event Structure

```
{
 "idListener": "integer",
 "email": "string"
}
```

The **Competitions** Microservice is subscribed to the "new-listener" event. Let's see then part of the **Competitions** sources. Let's start with their Web API:

Competitions - Retrieve Available Competitions

```
GET https://[host]/competitions

Successful response:
{
 "result": "success",
 "competitions": [
  {
    "inscriptions-end-date": "date",
    "inscriptions-start-date": "date",
    "description": "string",
    "rules": "string",
    "start-date": "date",
    "id": "integer"
  },
  {
    ...
  }
  ...
 ],
}

Error response:
{
 "result": "error",
 "message": "string"
}
```

Competitions - Register in a Competition

```
POST https://[host]/inscription
{
  "idCompetition": "integer",
  "idCompetitor": "integer"
}

Successful response:
{
 "result": "success"
}

Error response:
{
 "result": "error",
 "message": "string"
}
```

Competitions exposes two services. One returns all the competitions that are available for registration and another that performs the registration of a listener in a competition. Let's see how this Microservice is instantiated and started through its *Main* class:

Competitions Main

```
package main;

public class Main {

 public static void main(String[] args) {
  String connString = Objects.requireNonNull(
    System.getProperty("conn-string"),
    "specify conn-string as a jvm argument");
  String user = Objects.requireNonNull(System.getProperty("user"),
    "specify a database user as a jvm argument");
  String pass = Objects.requireNonNull(System.getProperty("pwd"),
    "specify a database pwd as a jvm argument");
  String pubsub = Objects
    .requireNonNullElse(System.getProperty("pub-sub-server"), "localhost");
  String pubsubPort = Objects
    .requireNonNullElse(System.getProperty("pub-sub-port"), "6379");

  // Redis event
  var redis = new RedisEvent(pubsub, Integer.valueOf(pubsubPort));
```

```
20
21    // Starting Web App ...
22    new Web(
23     new PublishInscriptionAdded(
24      new DefaultRadioProgram(
25        new JdbcCompetitionRepository(user, pass, connString)
26      ),
27      new JdbcListenerRepository(user, pass, connString),
28      redis
29     ),
30     7000
31    ).start();
32
33    // Starting subcription to new-listener event
34    new Thread("competition-subscription-to-redis") {
35     public void run() {
36      new NewListener(
37       new JdbcListenerRepository(user, pass, connString),redis)
38        .startListening();
39     }
40    }.start();
41   }
42  }
```

Note that *Main* instantiates the *Web* class and calls its *start*() method and after that instantiates and starts a new *Thread* which makes the subscription to the "new-listener" event (we'll come back to the subscription later). To instantiate the *Web* class we need the *PublishInscriptionAdded* dependency which is a decorator of *DefaultRadioProgram*, which is responsible for publishing the "inscription-added" event when a listener registers in a competition. Let's see next how this message is published:

Publishing inscription-added Message

```
1  package competition.model;
2
3  public class PublishInscriptionAdded implements RadioProgram {
4   private String EVENT_NAME = "inscription-added";
5   private RadioProgram radioProgram;
6   private Event event;
7   private ListenerRepository listener;
8
9   public PublishInscriptionAdded(RadioProgram radioProgram,
10    ListenerRepository listener, Event notification) {
11   this.radioProgram = radioProgram;
12   this.event = notification;
```

```
13    this.listener = listener;
14   }
15   ...
16
17   @Override
18   public void addInscription(int idCompetition, int idCompetitor) {
19    this.radioProgram.addInscription(idCompetition, idCompetitor);
20    event.publish(EVENT_NAME,
21      new InscriptionAdded(idCompetitor,
22        this.listener.listenerEmail(idCompetitor),
23        "Inscription Added Successfully",
24        "You have been enrolled successfully").toJson());
25   }
26  }
```

PublishInscriptionAdded is a decorator of *DefaultRadioProgram*. In the *addInscription*(...) method (line 18), it first delegates to enroll the listener (line 19) and then publishes the "inscription-added" event (line 20). Lets continue with the *InscriptionAdded* event class and their *json* structure:

InscriptionAdded Event

```
1  package competition.model;
2
3  public class InscriptionAdded {
4   private static final String MSG_BODY = "msg-body";
5   private static final String MSG_TITLE = "msg-title";
6   private static final String EMAIL_ADDRESS = "email";
7   private static final String ID = "id";
8   private int idCompetitor;
9   private String email;
10  private String msgTitle;
11  private String msgBody;
12
13  public InscriptionAdded(int idCompetitor, String email, String msgTitle,
14    String msgBody) {
15   this.idCompetitor = idCompetitor;
16   this.email = email;
17   this.msgTitle = msgTitle;
18   this.msgBody = msgBody;
19  }
20
21  public String toJson() {
22   return new Gson().toJson(Map.of(ID, idCompetitor, EMAIL_ADDRESS,
23     this.email, MSG_TITLE, this.msgTitle, MSG_BODY, this.msgBody));
```

```
 }
}
```

The *InscriptionAdded* class encapsulates everything necessary for another Microservice (**Notifications** in our case) to be able to send a notification. As we have done in the UserProfile Microservice, this class also provides a *toJson*() method to generate the following structure:

inscription-added Event Structure

```
{
 "id": "integer",
 "email": "string",
 "msg-title": "string",
 "msg-body": "string"
}
```

Note that in **Competitions** the listener's email is required to be added into the "InscriptionAdded" event. Because of that, in the *Main* class of the **Competitions** Microservice a new *Thread* is created with an instance of the Competitions.*model.NewListener* class which is responsible for *subscribing* to the "new-listener" event published by the **UserProfile** Microservice. Let's see how it is done:

Listening to new-listener Event

```
package competition.model;

public class NewListener implements OnMessageReceived {
 private ListenerRepository repository;
 private Event event;
 private String MSG_NAME = "new-listener";

 public NewListener(ListenerRepository repository, Event event) {
  this.repository = repository;
  this.event = event;
 }

 public void startListening() {
  this.event.listenTo(MSG_NAME, this);
 }

 @Override
 public void messageReceived(String message) {
  var event = new NewListenerEvent(message);
  this.repository.addNewListener(event.idListener(), event.email());
 }
}
```

model.NewListener.startListening() keeps the *thread* listening for the publication of the "new-listener" event (line 13 and 14). Furthermore, this instance of *NewListener* is passed as a parameter (second parameter) to the event, line 14, so that the publish/subscribe mechanism that implements the *Event.listenTo* abstraction notifies the *model.NewListener* through the *messageReceived*(...) method when a new message arrives. When received, the *messageReceived*(...) method is invoked by the publish/subscribe mechanism and the listener's email is persisted in the database of the **Competitions** Microservice.

Let's finally see how the *Web* class is instantiated and the services are exposed (similar to how the UserProfile Microservice does):

Competitions Web

```
1  package competition.web;
2
3  public class Web {
4
5   private RadioProgram radioProgram;
6   private int webPort;
7
8   public Web(RadioProgram radio, int webPort) {
9    this.radioProgram = radio;
10   this.webPort = webPort;
11  }
12
13  public void start() {
14   Javalin app = Javalin.create().start(this.webPort);
15   app.get("/competitions", allCompetitions());
16   app.post("/inscription", addInscription());
17
18   app.exception(RadioException.class, (e, ctx) -> {
19    ctx.json(Map.of("result", "error", "message", e.getMessage()));
20    // log error in a stream...
21   });
22
23   app.exception(Exception.class, (e, ctx) -> {
24    ctx.json(
25      Map.of("result", "error", "message", "Ups, somethong went wrong"));
26    e.printStackTrace();
27    // log error in a stream...
28   });
29  }
30
31  private Handler addInscription() {
32   return ctx -> {
```

```
    InscriptionData dto = ctx.bodyAsClass(InscriptionData.class);
    radioProgram.addInscription(dto.getIdCompetition(),
      dto.getIdCompetitor());
    ctx.json(Map.of("result", "success"));
   };
  }

  private Handler allCompetitions() {
   return ctx -> {
    var competitions = radioProgram.availableCompetitions();
    var list = new ArrayList<Map<String, String>>();

    for (RadioCompetition c : competitions) {
     var toJson = new HashMap<String, String>();
     toJson.put("id", String.valueOf(c.id()));
     toJson.put("description", c.description());
     toJson.put("rules", c.rules());
     toJson.put("inscriptions-start-date",
       c.inscriptionStartDate().toString());
     toJson.put("inscriptions-end-date", c.inscriptionEndDate().toString());
     toJson.put("start-date", c.startDate().toString());
     list.add(toJson);
    }
    ctx.json(Map.of("result", "success", "competitions", list));
   };
  }
}
```

Finally, we have the **Notifications** Microservice that does not expose *https* services but does subscribe to the "inscription-added" event. Let's see how the dependencies are instantiated and injected in *Main* class:

Notifications Main

```
package main;

public class Main {
 public static void main(String[] args) {
  String pubsub = Objects
    .requireNonNullElse(System.getProperty("pub-sub-server"), "localhost");
  String pubsubPort = Objects
    .requireNonNullElse(System.getProperty("pub-sub-port"), "6379");

  new EmailDispatcher(
```

```
11      new RedisSubscriber(pubsub, Integer.valueOf(pubsubPort)),
12      new EmailVendorProvider())
13    .startListening();
14  }
15 }
```

model.EmailDispatcher (line 10) is the class who listens to the "inscription-added" event, let's see how its implementation:

Notifications EmailDispatcher class

```
1  package notifications.model;
2
3  public class EmailDispatcher implements OnMessageReceived {
4   private Event listenToEvent;
5   private EmailProvider provider;
6
7   public EmailDispatcher(Event event, EmailProvider provider) {
8    this.listenToEvent = event;
9    this.provider = provider;
10  }
11
12  public void startListening() {
13   listenToEvent.listenTo(this);
14  }
15  @Override
16  public void messageReceived(String message) {
17   var msg = new InscriptionAdded(message);
18   this.provider.send(msg.email(), msg.title(), msg.body());
19  }
20 }
```

Note that in a similar way to **Competitions**, *EmailDispatcher.startListening*(), keeps listening for "inscription-added" events. *this* instance of *EmailDispatcher* is passed as a parameter (line 13) in order to be notified via the *messageReceived*(. ..) method (line 16) when the "inscription-added" event is received. When that happens, an email is sent (line 18).

With this we finish the explanation of the implementation. Source code of all the Microservices implemented can be found in github/competitions*, github/userprofile† y github/notifications‡.

Let's see finally how the container diagram of RadioA with Microservices architecture looks like:

*https://github.com/enriquemolinari/microservice-competitions
†https://github.com/enriquemolinari/microservice-userprofile
‡https://github.com/enriquemolinari/microservice-notifications

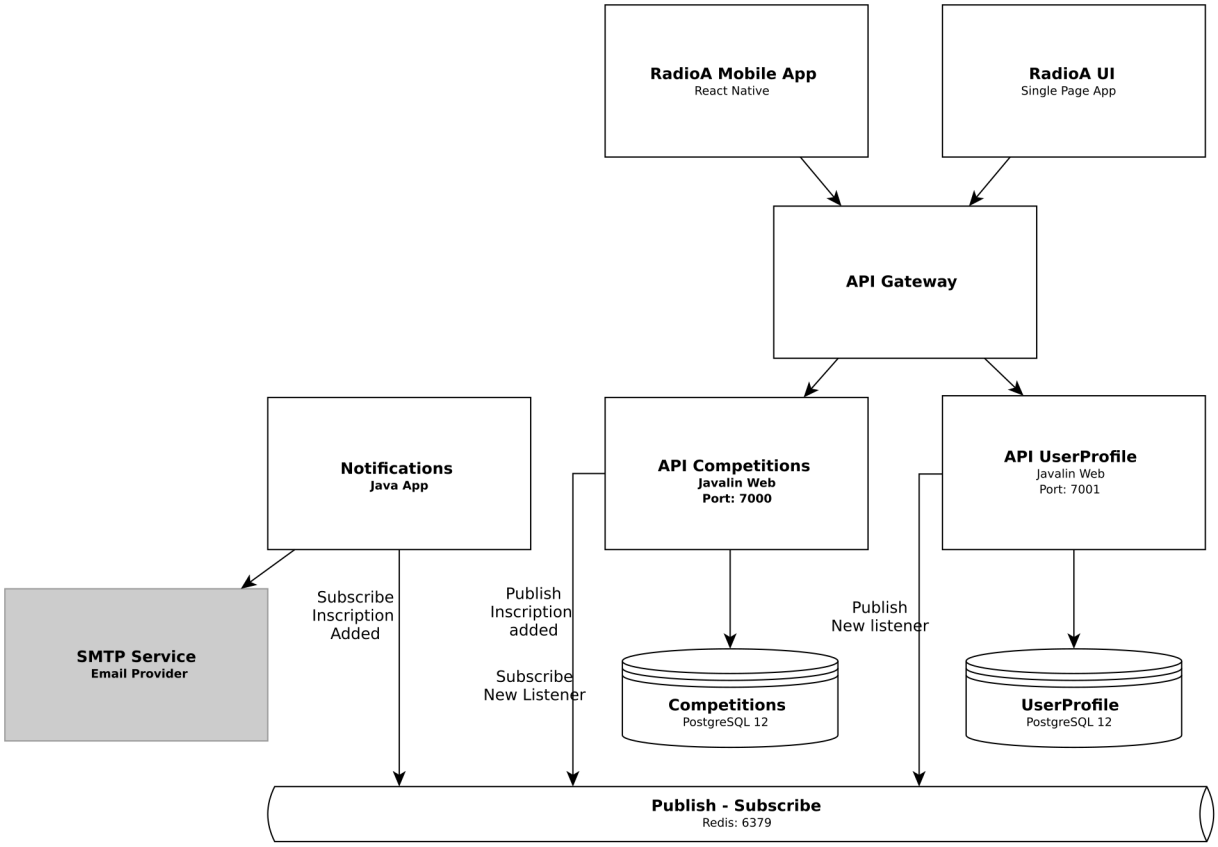

Figure 4.8: Microservices: Container Diagram

Some Comments about the Implementation

Let's review some important decisions regarding the implementation we just have done:

Duplicated Code: You may have noticed that there is duplicated code between the Microservices. As we mentioned earlier, sharing reusable modules between Microservices creates coupling between them, and one of the main architectural rules of this style is precisely being independent. On the other hand, this style makes sense where there is a significant staff of developers divided into teams, each responsible for a Microservice, and where code reuse occurs mainly within each team and not between teams.

Publish/Subscribe Mechanism: We have used Redis Pub/Sub* as an implementation of the publish/-subscribe mechanism. It is really very simple to use, extremely fast, and for certain requirements it could be sufficient. However, it is not the right implementation for Microservices where messages cannot be lost. For example, if a Microservice is not listening when messages are published, when reconnecting to the publish/subscribe mechanism, it should be able to consume those messages already sent. This is known as *durability*. Let's look at some essential features to keep in mind when selecting a publish/subscribe technology:

*https://redis.io/topics/pubsub

1. **Durability**: Do we want messages to remain available from being consumed until they are actually consumed by all subscribers?
2. **Persistence**: Do we want messages to persist so they survive a crash?
3. **Failure on the Consumer**: If for some reason the consumer receives the message, but crashes before performing an action with it, we should be able to mark that message as not done somehow.
4. **Message Order**: Do we need to preserve the order of the messages for consumption?

Security: The **UserProfile** authentication service generates an UUID* after authentication succeeds. It would be better to generate a JWT† token, as it is more secure. Every request, before being served by the Microservices, goes through an API Gateway who checks if the token is valid. If it is, it forwards the request to the Microservice. The container diagram in figure 4.8 shows that it is the API Gateway who receives the requests from clients to forward them to the corresponding Microservice.

Coupling between Microservices: The only coupling between Microservices is the structure of the messages. Here it is necessary for the teams to agree on its definition and be backwards compatible when making changes.

Operations: One of the most important complexities generated by this style of architecture has to do with the management that must be carried out in production. Effectively monitoring an application with Microservices architecture is not a simple task. Even more complex if we have a different technology stack in each Microservice. To alleviate this, Microservices do tend to share libraries or services, for example, for logging, whose entries end up in a centralized and easy-to-review repository. In general, these guidelines are defined by the operations team.

Database Changes and Event Publication Atomically: Both Microservices **UserProfile** and **Competitions** requires to do changes in their database and publish an event and that must be done atomically. For example, in the **UserProfile** Microservice after we do the creation of a new listener (an insert in the database) we have to do the publication of the "new-listener" event and must be atomically. The same happens in the **Competitions** Microservice which after we register a listener in a competitions we have to publish the "inscription-added" event. As we mentioned earlier, one possibility is to implement the outbox pattern. Another option, could be the use tools like Debezium‡, that monitor the transaction logs of databases and publish events for each *insert*, *update* or *delete* in an atomic way.

Modular Architecture vs Microservices

Although it seems that Microservices are the only option to spaghetti code, as we have seen in this book, this is **not the case**. What is necessary to avoid spaghetti code is to define a solid architecture with clear rules, validated in some way (in compilation or with automated tests) on every commit.

*https://en.wikipedia.org/wiki/Universally_unique_identifier
†https://jwt.io/
‡https://debezium.io/

It is very clear that if we cannot create a layered, hexagonal and/or modular architecture and we go to Microservices, each Microservice will end up in spaghetti code. If, on the other hand, I have an application with vertical modules, such as the one we built in the previous chapter, going to Microservices is much simpler than if we want to go to Microservices starting from a monolithic application with spaghetti code.

I consider that starting the development of an application with a modular architecture with well-defined rules is better than starting directly with Microservices. This is recommended by Fowler in his article Monolith First*.

It is also possible to have asynchronous communication between modules of a monolithic application. The fact of having a modular monolith does not prevent each other from communicating via a publish/subscribe mechanism. As we have already mentioned, a solid modular architecture does not require the help of mechanisms like the one that Java9 gives us. It can be implemented simply with packages or namespaces as long as clear rules are defined regarding the dependencies between them, their criteria for grouping classes in them, and most importantly, where these rules are validated in every commit.

Notes

50 Fundamentals of Software Architecture: An Engineering Approach 1st Edición. Mark Richards, Neal Ford. 2020.

51 Modular Monoliths.

52 Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Martin Kleppmann. 2017.

53 Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. Martin Kleppmann. 2017.

54 Distributed Systems for practitioners. Dimos Raptis. 2020.

55 Sagas. H. Garcia-Molina and K. Salem. Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, 1987.

56 Distributed Systems for practitioners. Dimos Raptis. 2020.

57 Building Microservices: Designing Fine-Grained Systems. Sam Newman. 2015.

58 Domain-Driven Design: Tackling Complexity in the Heart of Software, Eric Evans. 2003.

59 Domain-Driven Design: Tackling Complexity in the Heart of Software, Eric Evans. 2003.

60 Building Microservices: Designing Fine-Grained Systems. Sam Newman. 2015.

61 Fundamentals of Software Architecture: An Engineering Approach 1st Edición. Mark Richards, Neal Ford. 2020.

*https://martinfowler.com/bliki/MonolithFirst.html

III Quality Attributes

In the previous two chapters we talked almost completely about the **Maintainability** quality attribute. We structure an application using different criteria and style with the main goal of reducing the complexity of the software product. This allows us to increase the speed of the changes that every software product with relative success has constantly. Different structures and styles of architecture also favor other quality attributes, not just maintainability. For example, Microservices favor, in addition to modifiability, **availability**. When one Microservice is down, the rest of the application continues working.

How do we define quality attributes?

In the book Software Architecture in Practice[62] quality attribute is defined as a measurable and testable property of the system that is used to indicate how well the system satisfies the needs of its stakeholders.

Quality attributes are also called non-functional requirements. In the recent book Fundamentals of Software Architecture[63], the authors incorporate a third way of calling them: *architecture characteristics*. They say that it best describes how important they are to the success of the architecture.

In this book I refer to them as quality attributes because that is how you will find it in most of the software architecture literature. In any case, all the authors agree that these refer to requirements not related to the problem domain, but to technical needs of equal or greater importance. Many times we re-design a system because the performance is bad, or because the availability is not enough or because we need to improve the scalability while maintaining the functionality in the same way. This demonstrates the importance of quality attributes for the development of a system.

In the definition of quality attributes cited above, it is mentioned that each attribute must be measurable and testable. It is not enough to say that a system is "modifiable". Every system supports certain changes better than others, and in the same way it happens with other attributes. For example, a system can support better the failure of a particular container but not the others. This requires an extensive specification of quality attribute requirements, as can be found in the book Software Architecture in Practice[64].

We have already discussed modifiability in detail in the previous chapters. In this part of the book we focus on detailing the attributes **availability** and **scalability**.

Notes

62 Software Architecture in Practice. Third Edition. Len Bass, Paul Clements, Rick Kazman. 2012.

63 Fundamentals of Software Architecture: An Engineering Approach 1st Edición. Mark Richards, Neal Ford. 2020.

64 Software Architecture in Practice. Third Edition. Len Bass, Paul Clements, Rick Kazman. 2012.

5. Availability

Introduction

You have probably heard that a certain system has **High Availability** or its availability is five 9s (99.999%). But what does this mean? What is availability? Can it be calculated? Let's start by answering these questions.

If a system upon receiving a request responds according to what was specified and implemented, then it is available. Since the causes that could occur to prevent a system from being available are not possible to know in advance, the availability of a system is defined as the **probability** that the system is in operation at a given time. From this it follows the reason for the **notation** of the **nines** to specify the availability of a system. If I say that a system has 99.9% availability, what I am really saying is that in a year it could be 8.76 hours unavailable. How do I get to that result? 99.9 / 100 = 0.999. Multiplying 365 * 0.999 = 364,635, I get the number of days in the year that the system will be available. In other words, it will not be available 365 - 364,635 = 0.365 days in the year. If we do 0.365 * 24 we get 8.76 hours a year not available. Performing these same calculations, according to the number of nines, we are able to know the total unavailable time of a system, as shown in the following table:

Availability	9s	Downtime
90%	one	36,5 days/year
99%	two	3,65 days/year
99.9%	three	8,76 hours/year
99.99%	four	52 minutes/year
99.999%	five	5 minutes/year
99.9999%	six	31 seconds/year

High availability has those systems that have an availability of five nines or more. Now it remains for us to understand how we calculate the availability of an element of the system to later be able to calculate the availability of the whole system.

Calculating the Availability

The availability of an element of a system is calculated using its average time between two failures (MTBF) and its average time taken to repair it after a failure (MTTR), using the following formula:

$$\text{availability} = \frac{MTBF}{MTBF+MTTR}$$

From this formula we can see that the higher the MTBF, the more availability we will have. In the

same way, the faster we can get out of a failure, the better availability we will have. MTBF and MTTR are represented in hours.

How do I know what the MTBF is for an element in a system?

For **hardware elements** the MTBF can be obtained from the manufacturer and the MTTR from our support and infrastructure team or from the provider with whom we have contracted the service. Hardware failures usually require the component to be replaced. There are several factors that go into understanding the MTTR in hardware. Replacing a damaged network card or other network element is not the same as replacing a damaged disk. In the latter case, it will depend, for example, on whether we have data replicas ready to be used or we have to restore a backup. In addition, the moment in which it happens also influences, if it is within normal working hours or if it is on a weekend. It will depend on how we are organized for such events. To give an example, the MTTR value in KLM[65] is 4 hours.

Hardware failures are usually described by the bath tub[66] curve. That is, at the beginning of the life of a hardware element the chances of failure are usually high, then during the normal life cycle the probability of failure is very low, and at the end of its useful life the probability of failure is high due to degradation of their characteristics. As illustrated in figure 5.1.

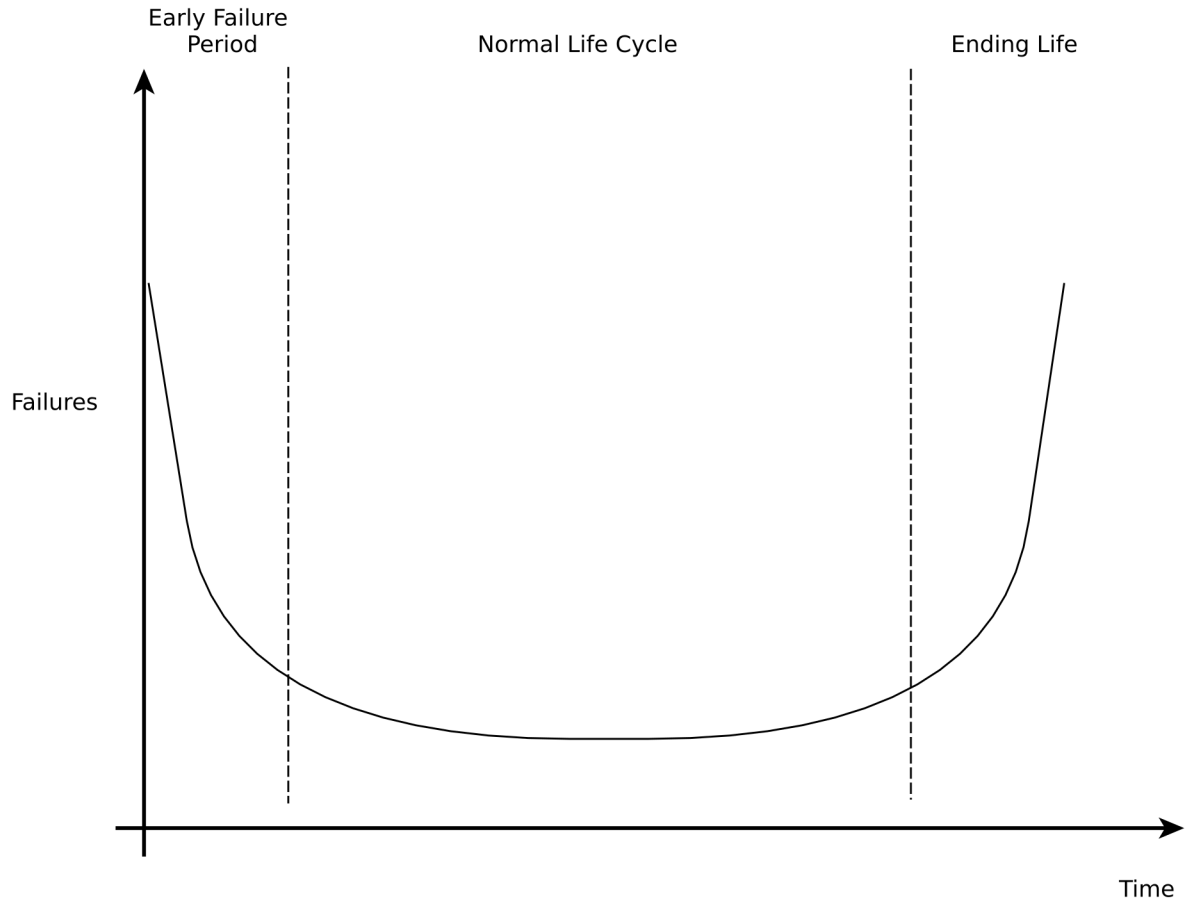

Figure 5.1: Bath Tub

The MTBF calculation for **software elements** has multiple factors. Like its complexity, size (KLOC), experience of the development teams and time spent on testing, to mention only the most important ones. To perform this calculation, it is necessary to observe the behavior of the application during a given period of time, measuring its failures. Which is not easy especially in applications with some success since it has a high frequency of changes. The MTTR is generally the time it takes to restart the application, which is generally low. Cisco[67], for example, assumes 10,000 hours of MTBF for products with recent changes or new functionality, and 30,000 hours of MTBF for stable products. And in both cases it assumes an MTTR of 0.1 hours (6 minutes to restart).

Being able to obtain the availability of each element of a system, we can calculate its **total availability**. Suppose the deployment diagram in figure 5.2.

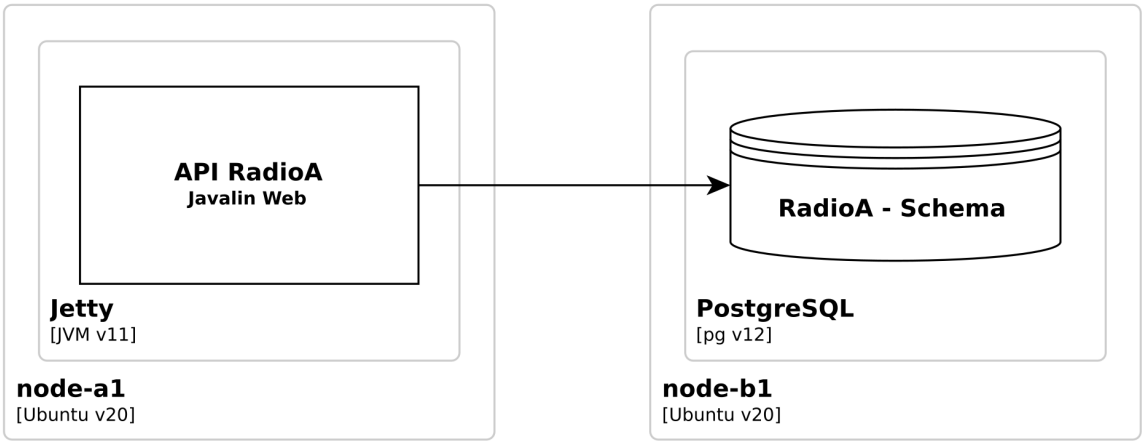

Figure 5.2: RadioA API Web - Deployment Diagram

Figure 5.2 shows the RadioA API application deployed on a server with Ubuntu 20, running a Java virtual machine version 11 with Jetty*. Also, the database is deployed on another server with the same operating system and running PostgreSQL version 12.

To calculate the availability of the system, we first have to understand the availability of each of the elements involved. It is important to mention that there are many more elements than those shown in the deployment diagram. For example, in a server we have the motherboard, the processor, the network card and storage devices, just to mention a few. We also have network devices (switch, router, etc.) that allows us to connect the application with the database. All these elements must be taken into account. To simplify this demonstration we will take into account the elements shown in the diagram.

Let's start by calculating the availability of each element :

$$\textbf{Application Server (node-a1)} = \frac{50000}{50000 + 8} = 0,999840026$$

(Assuming 50.000 hours of MTBF and 8 hours of MTTR).

$$\textbf{Ubuntu v20 (node-a1)} = \frac{30000}{30000 + 0.1} = 0,999996667$$

(Assuming 30.000 hours of MTBF and 0.1 hours of MTTR).

$$\textbf{Jetty/JVM v11 (node-a1)} = \frac{30000}{30000 + 0.1} = 0,999996667$$

(Assuming 30.000 hours of MTBF and 0.1 hours of MTTR).

*https://www.eclipse.org/jetty/

RadioA (Java Swing App) $= \frac{20000}{20000 + 0.1} = 0,999995$

(Assuming 20.000 hours of MTBF and 0.1 hours of MTTR).

Database Server (node-b1) $= \frac{50000}{50000 + 8} = 0,999840026$

(Assuming 50.000 hours of MTBF and 8 hours of MTTR).

Ubuntu v20 (node-b1) $= \frac{30000}{30000 + 0.1} = 0,999996667$

(Assuming 30.000 hours of MTBF and 0.1 hours of MTTR).

PostgreSQL v12 (node-b1) $= \frac{30000}{30000 + 0.1} = 0,999996667$

(Assuming 30.000 hours of MTBF and 0.1 hours of MTTR).

Up to here we have the availability of each element of the system. How does the availability of each individual element impact the total availability of the system?

To answer this question we have to understand what happens to the system if any of the elements fails and become unavailable. With the deployment diagram presented in figure 5.2, we can see that if any of these elements fails, the whole system **will be unavailable** to the end user. Therefore, in configurations of this type, *total availability* is calculated by multiplying the availability of each of its elements, as we can see in the following formula, called **serial configuration**:

$$\textbf{Serial} = \prod_{i=1}^{n} availabilityOfElement_i$$

Where n is the total number of elements.

Applying the formula we have the following:

$$\begin{aligned} \textbf{Availability (app server)} &= 0,999840026 \text{ (node-a1)} \\ &\times\ 0,999996667 \text{ (Ubuntu v20)} \\ &\times\ 0,999996667 \text{ (Jetty/JVM v11)} \\ &\times\ 0,999995 \text{ (RadioA App)} \\ &=\ 0,999828362 \end{aligned}$$

$$\begin{aligned}\textbf{Availability (db server)} &= 0,999840026 \text{ (node-b1)} \\ &\times\ 0,999996667 \text{ (Ubuntu v20)} \\ &\times\ 0,999996667 \text{ (PostgreSQL)} \\ &=\ 0,999833361\end{aligned}$$

$$\begin{aligned}\textbf{System Availability} &= 0,999828362 \text{ (app server)} \\ &\times\ 0,999833361 \text{ (db server)} \\ &=\ 0,999661752 \times\ 100\ =\ 99,97\%\end{aligned}$$

Note that even though we are optimistic about the MTBF and MTTR assigned to each element of the system and without including other elements such as the network card, processor, etc., we obtain an availability of three nines (8.76 hours per year without service). While not bad at all and probably sufficient for many businesses, we don't get high availability.

The only way to have *high availability* is by eliminating what is known as **Single Point of Failure**. A single point of failure is an element of the system that if it fails, the whole system will fail. The only way to do it is by replicating these elements (removing from the equation that if the application has bugs, the replicas will not help). This has a high cost in time and money. So it is important to understand if it is really necessary. If our MTTR is relatively low (or low enough for our business), it is probably not necessary to completely eliminate single points of failure.

If we incorporate replicas, the availability calculation is different, since for the system to fail all replicas would have to fail. The calculation formula is called **parallel configuration** and is shown below:

$$\textbf{Parallel} = 1 - [\prod_{i=1}^{n}(1 - availabilityOfElement_i)]$$

Where n is the total number of elements in parallel.

So, suppose we deploy the same application from the previous example, but replicating the application server (node-a1), as figure 5.3 shows below:

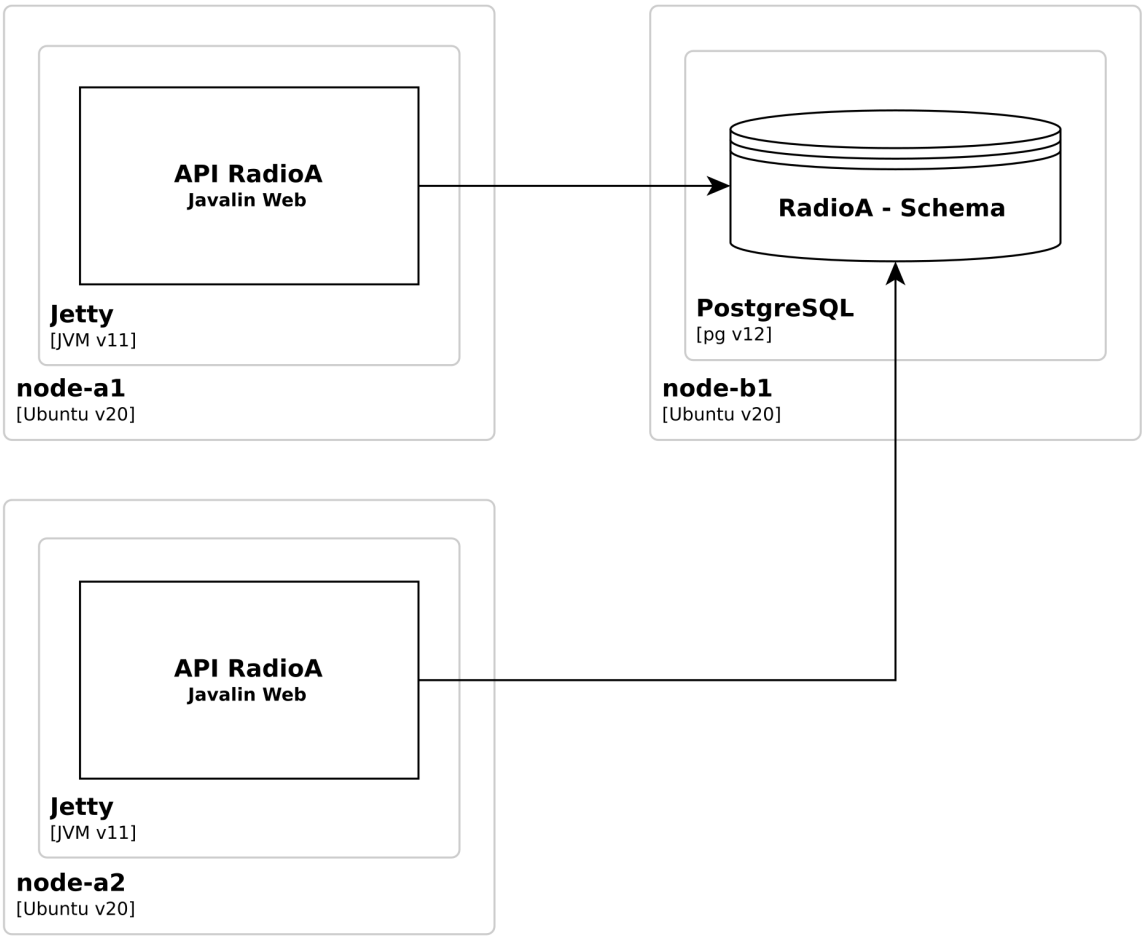

Figura 5.3: RadioA API Web - Deployment Diagram

Let us now calculate the availability of the system shown in figure 5.3. We take the availability of the application server (app server) that we calculated before and use it now in the parallel configuration formula:

$$\begin{aligned}\textbf{Availability (app server)} &= 1 - (1 - 0,999828362) \times (1 - 0,999828362) \\ &= 1 - (0,000171638 \times 0,000171638) \\ &= 1 - 0,000000029 \\ &= 0,999999971 \times 100 = 99,99999\%\end{aligned}$$

Let's observe how the replication of the app server has increased the availability of it. Now we take the availability of the database server (bd server) that we have already calculated before and finally obtain the system availability by using the serial configuration formula:

$$\begin{aligned}\textbf{System Availability} &= 0,999999971\ (app\ server) \\ &\times 0,999833361\ (db\ server) \\ &= 0,999833361 \times 100 = 99,98\%\end{aligned}$$

Note that although there is an improvement, this improvement is not significant yet. Since the database server is still a single point of failure, the serial formula brings the total system availability closer to the value of the element with less availability. Which sounds reasonable given that if that element fails, the entire system fails. So, the entire system tends to have the availability of the element with less availability.

Finally let's move on to deploying RadioA replicating the database server as well. Figure 5.4 shows a master/slave replication of the database working as an active/passive model. That is the passive database server (node-b2) becomes active in case of the current active database fails.

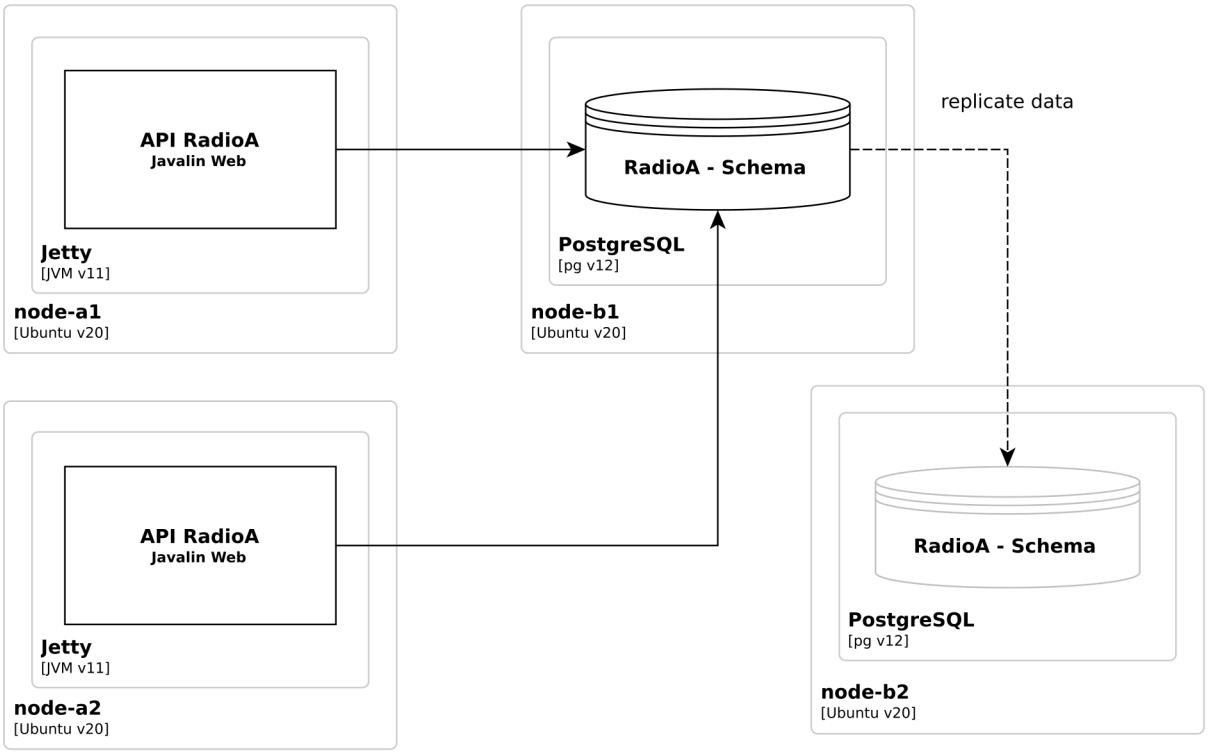

Figure 5.4: RadioA API Web - Deployment Diagram

Let's now calculate the availability of the database server with the parallel configuration formula:

$$\begin{aligned}\textbf{Availability (bd server)} &= 1 - (1 - 0,999833361) \times (1 - 0,999833361) \\ &= 1 - (0,000166639 \times 0,000166639) \\ &= 1 - 0,000000028 \\ &= 0,999999972 \times 100 = 99,99999\%\end{aligned}$$

And now then we calculate the system availability with the configuration of figure 5.4:

$$\begin{aligned}\textbf{System Availability} &= 0,999999971\ (app\ server) \\ &\times 0,999999972\ (db\ server) \\ &= 0,999999943 \times 100 = 99,99999\%\end{aligned}$$

It is only with all the elements replicated where we achieve high availability. On the other hand, it is important to mention that when we replicate elements, it is necessary to introduce an additional element in the system that balances the requests between the replicated elements, in the case of replicas where both are active (like the app server in previous diagrams). In the case of active/passive models, as is the case of the database server in figure 5.4, there should be a mechanism that, when the primary/active server fails, all the requests get redirected to the secondary/passive server. This mechanism is known as failover*. Any of these mechanisms used becomes part of the system, which must also be included in the calculation of availability, which contribute to reduce it (unless we also replicate them).

As you may have noticed, the cost of having high availability is high. This is why it is important to understand how necessary it is. An intermediate option is to replicate only those elements of the system that have a high MTTR, since if the MTTR is low, the availability does not decrease as much.

Other Issues Affecting Availability

There are others two factors that affect the availability of a system. One is the **environment**, for instance, power outages, fires, earthquakes and other similar events. To have a low MTTR in these cases, it is required to replicate the entire data center in a remote location. The other factor that affects availability more frequently are **human errors**. When we carry out any type of software upgrade, reconfigure servers or network equipment, or deploy new versions of applications, there is a risk of making mistakes. Automating manual tasks helps to decrease human errors. Perform the actions in low environments before doing it in production helps too.

Finally, to be prepared for a possible failure during new application releases to production, it is extremely important to design the systems in such a way that it is possible to go back to the previous version in a simple way. If every time we deploy a new version of the system in production it is possible to go back to the previous version, then the MTTR will always be low in these cases.

*https://en.wikipedia.org/wiki/Failover

Although this is not always possible, making this effort saves some headaches. To keep backward compatibility with previous and new versions of a system we should always iterate the database schema adding new tables, columns, etc, but not removing or changing existing ones for instance. If database changes for a new version of an application are backward compatible, we should deploy them in production before the new version of the application. Then, when moving the new version of the application to production, if we find problems, we can go back to the previous version in a simple way. Once the new version of the application has been in production for some weeks, only then we could apply scripts to delete unused tables or columns or add additional restrictions (not-nulls for example) to the database.

Availability in Microservices

Microservice architecture, given their asynchronous nature, could have increased availability depending on how the business sees it. That is, suppose we deploy the RadioA application with Microservice architecture as we saw in the previous chapter, without any replicated element. And suppose the **Notifications** Microservice crash. Since the mobile and web application are still available to the end user, it can be considered that the system is still available (if the business considers that a delay in sending emails does not count as a complete failure of the system). With this we could say that the total availability of the system will be the availability of the Competitions and UserProfile service (without incorporating the Notifications service into the equation).

Notes

65 Calculating Total System Availability. Hoda Rohani, Azad Kamali Roosta

66 Reliability and availability basics.

67 High Availability Network Fundamentals. Chris Oggerino. 2001.

6. Scalability

Scalability is the ability of a system to handle a greater demand without decreasing performance. There are two ways to scale: **vertical** or **horizontal**. We scale vertically (or also known as scale-up) when we improve the hardware where the application runs, that is, we increase the processing capacity, incorporate faster disks and increase the memory. We scale horizontally (or scale-out) by adding more computers with similar processing capacity, distributing the load among them.

Scaling horizontally is the only way to handle the demand for the so-called "planet-size" applications (Amazon, Google, Netflix, etc). But also, for a large number of organizations, scaling-out might be the only possible way due to the costs of upgrading hardware.

In general, when the performance of an application decreases we can always find a bottleneck. By doing performance and capacity tests, using tools like JMeter*, we would be able to find where the bottleneck is in an application. It is important to determine the bottleneck and verify that the performance problem is not caused by a bug in the application or some inefficient algorithm or database query. If we conclude that we simply need more resources then we must determine whether we need them at the application level or in the database. In general, the bottlenecks are usually in databases and unfortunately scaling a database is more complex than scaling an application as we will see below.

Scaling Web Applications

Before adding resources we must verify that the bottleneck in an application is not due to the container where it is running. That is, we must check that the web container cannot be configured to allow a greater number of concurrent requests, for example. Perhaps, with a simple configuration change, we do not need to add more resources in the system, which increases the cost and time of implementation and maintenance.

We build Web applications that we could say belong to one of the following two categories: **stateless** or **stateful**. Stateless Web applications can respond to a request using only the information that comes in it, without the need to remember any previous request. While stateful Web applications need to store certain information from a previous request in order to continue processing new requests. In general, this information is stored in what are called *session* variables, which by default and depending on the container used, are saved in memory or on the local disk of the application server. These session variables are generally created when the user is authenticated in the system and are destroyed when logging out.

Assuming that the application is stateful, when scaling by adding new application servers, it would be necessary to track in which of the servers the session for each user was created. Because if the

*https://jmeter.apache.org/

session is created on a server but the next request is sent to another server, the system will think that the user is not authenticated yet sending him to the login screen again. This is solved by configuring a load balancer with the mechanism called *sticky session* or *session affinity*. This mechanism registers each session identifier with the server where it is created, in order to redirect users to the servers where they have created the session.

Another way to scale stateful web applications, and avoid introducing affinity mechanisms, is to move session storage away from application servers. The great number of NoSQL databases (mostly open source) make this the best solution today. A database of category key/value*, in memory and with high availability (deployed in a cluster) is an excellent solution to store session data. Figure 6.1 shows a deployment diagram with two application servers using a Redis† cluster where the session data is stored.

*https://en.wikipedia.org/wiki/Key%E2%80%93value_database
†https://redis.io/

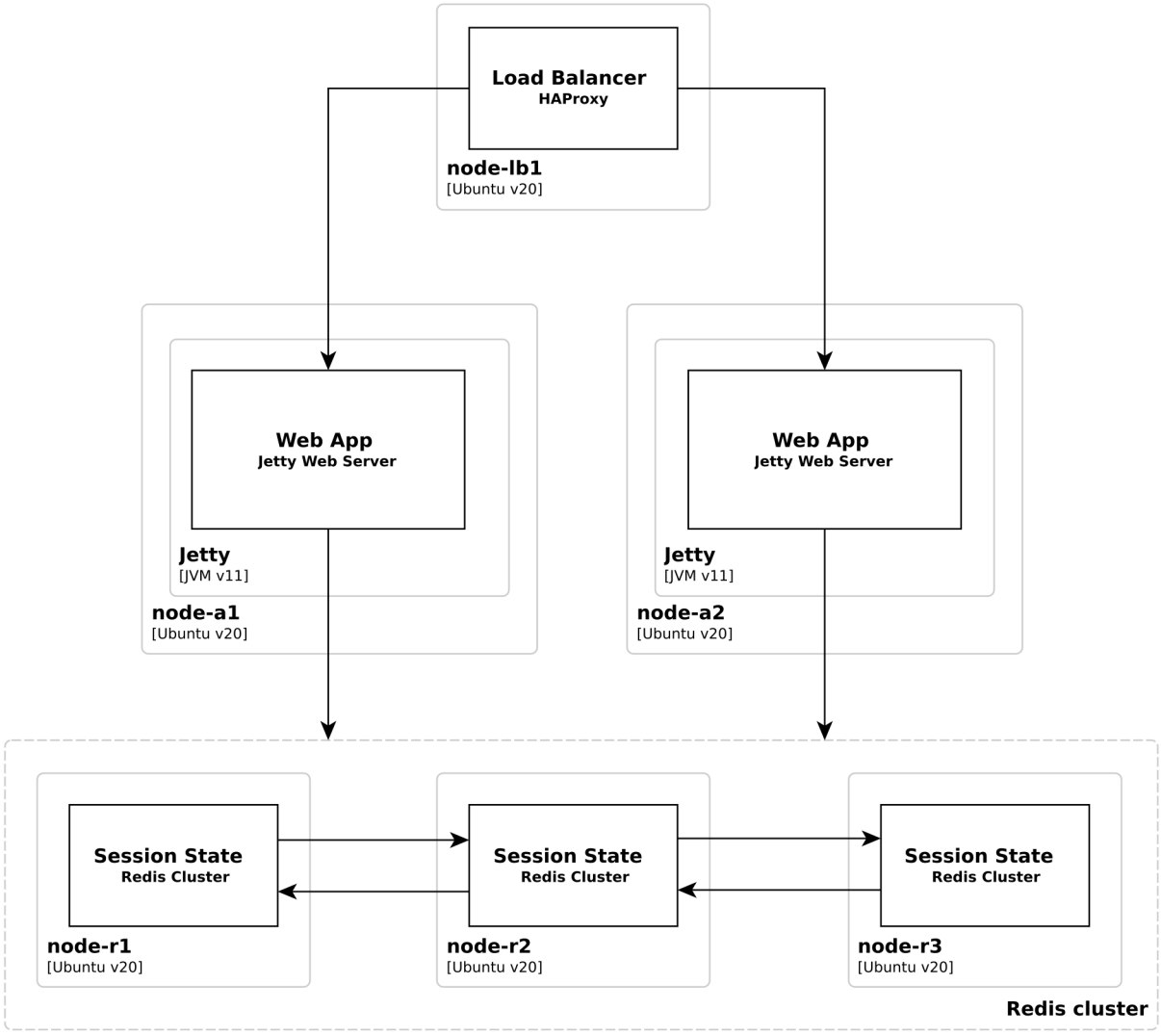

Figure 6.1: Scaling Stateful Web Applications - Deployment Diagram

Figure 6.2 shows an stateless Web application deployment.

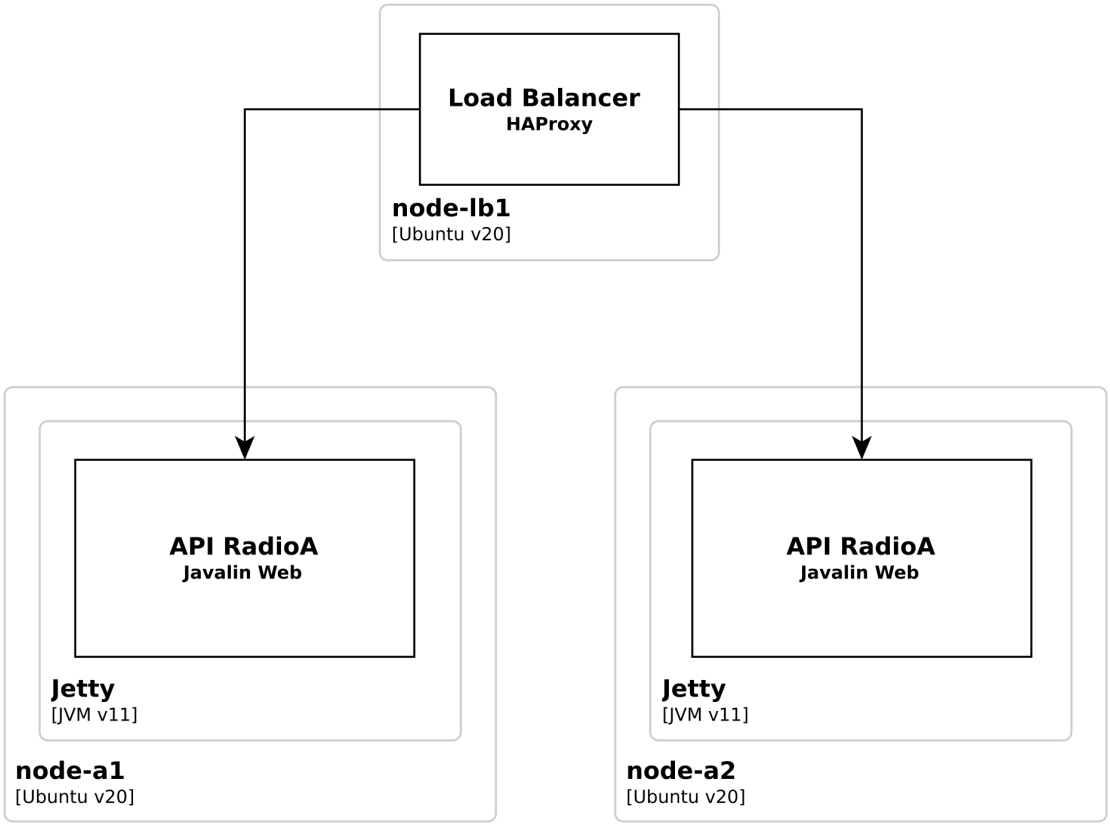

Figure 6.2: Scaling Stateless Web Applications - Deployment Diagram

In both cases the way to scale is simple, just add more application servers and more Redis instances in the case of a stateful deployment.

Scaling Databases

Scaling databases is complex mainly because we have to distribute the data on more than one computer, which leads us to have to deal with the complexities of distributed systems mentioned before. Relational databases and their foundational properties, such as ACID transactions, were designed to operate on a single centralized computer in a moment where there was no thought to requiring capacity for the demand imposed by "planet-size" applications. Companies like Google and Amazon have had to design their own databases, essentially distributed, to be able to scale and respond to their demand on a global scale. Thus the first NoSQL databases were born.

So, we could say that relational databases were designed to guarantee consistency and atomicity, among other things. They provide us with simple mechanisms to use so that as developers we don't have to deal with it. However, they do not provide us with simple mechanisms to have high availability or to scale. On the other hand, NoSQL databases do give us the ability to have high

availability and scale without too much effort, they were designed for this. But, they do it at the cost of consistency.

Let's start then by analyzing essential issues of distributed databases and then continue looking at different options that allow us to scale the databases.

CAP and PACELC Theorems

We said before that we could achieve high availability and scalability by distributing the data. And we also said that to achieve this we have to sacrifice consistency. The CAP (and PACELC) theorem explains why this happens. The CAP Theorem suggests that in a **network outage** we have to decide between offering consistency or availability (but not both). CAP represent the following properties:

Consistency: Each read will receive the value of the most recent write operation.

Availability: Every request gets a (non-error) response regardless of the individual state of a node.

Partition Tolerance: The system does not fail, regardless of messages being dropped or delayed between nodes caused by a network outage.

As shown in figure 6.3, suppose two databases, A and B, connected via a network where both have a record *X == 50* stored. At some point, the network fails and the connection between the databases is interrupted. Imagine then that in this state, a write request arrives at database A, in order to assign *100* to *X* and a read request arrives to database B for the *X* value. This is where the system has two options: perform both requests in A and B, returning an old value of X, which provides availability but not **consistency** or return an error providing consistency due to the new value of *X* cannot go from A to B, losing **availability**.

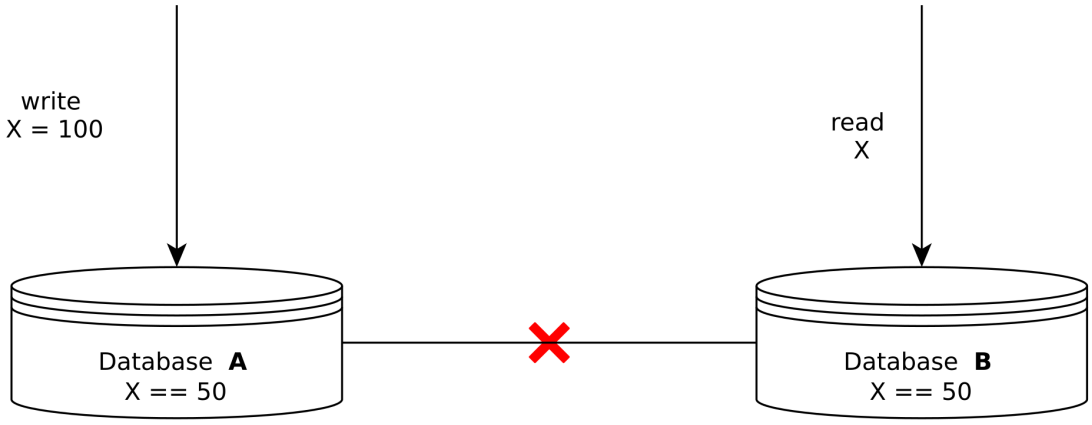

Figure 6.3: CAP Theorem

Thus, distributed databases are usually classified as **CP** or **AP**, depending on which property they guarantee in the event of a network failure.

It is important to understand that the choice between being a CP or an AP is relevant only when there is a network outage, which does not happen frequently. As long as the system is operating normally, both properties (Consistency and Availability) could be guaranteed. Because of this, there is an extension to the CAP theorem called **PACELC**[68], which says the following:

1. Where there are network outages (**P**), we have to decide if we want to provide availability (**A**) or provide consistency (**C**).
2. But when the system is running normally (**E**lse), one has to choose between low **L**atency and **C**onsistency.

That is, normally (in the absence of network partitions) write operations can be delayed until the changes are replicated throughout the system, having higher latency but guaranteeing consistency (EC). Or have minimal latency at the cost of not being consistent (EL).

In this way, distributed databases can be cataloged in the following ways:

AP/**EL**: Availability (A) over consistency if there are network partitions (P) and low latency in normal situations (EL).

AP/**EC**: Availability (A) over consistency if there are network partitions (P) and consistency, delaying writes in normal situations (EC).

CP/**EL**: Consistency (C) over availability if there are network partitions (P) and low latency in normal situations (EL). This configuration probably is not used really.

CP/**EC**: Consistency (C) over availability if there are network partitions (P) and consistency, delaying writes in normal situations (EC).

Scaling Reads

In general, applications receive a greater number of read operations than writes. So, if we detect that the bottleneck is in the database (rather than on the application), the easiest way to scale a database is scaling read operations.

The vast majority of databases have replication mechanisms built-in. Which would allow us to have a database to handle only write operations and others to handle read operations. In this way we distribute the load in several database servers. This is illustrated in Figure 6.4.

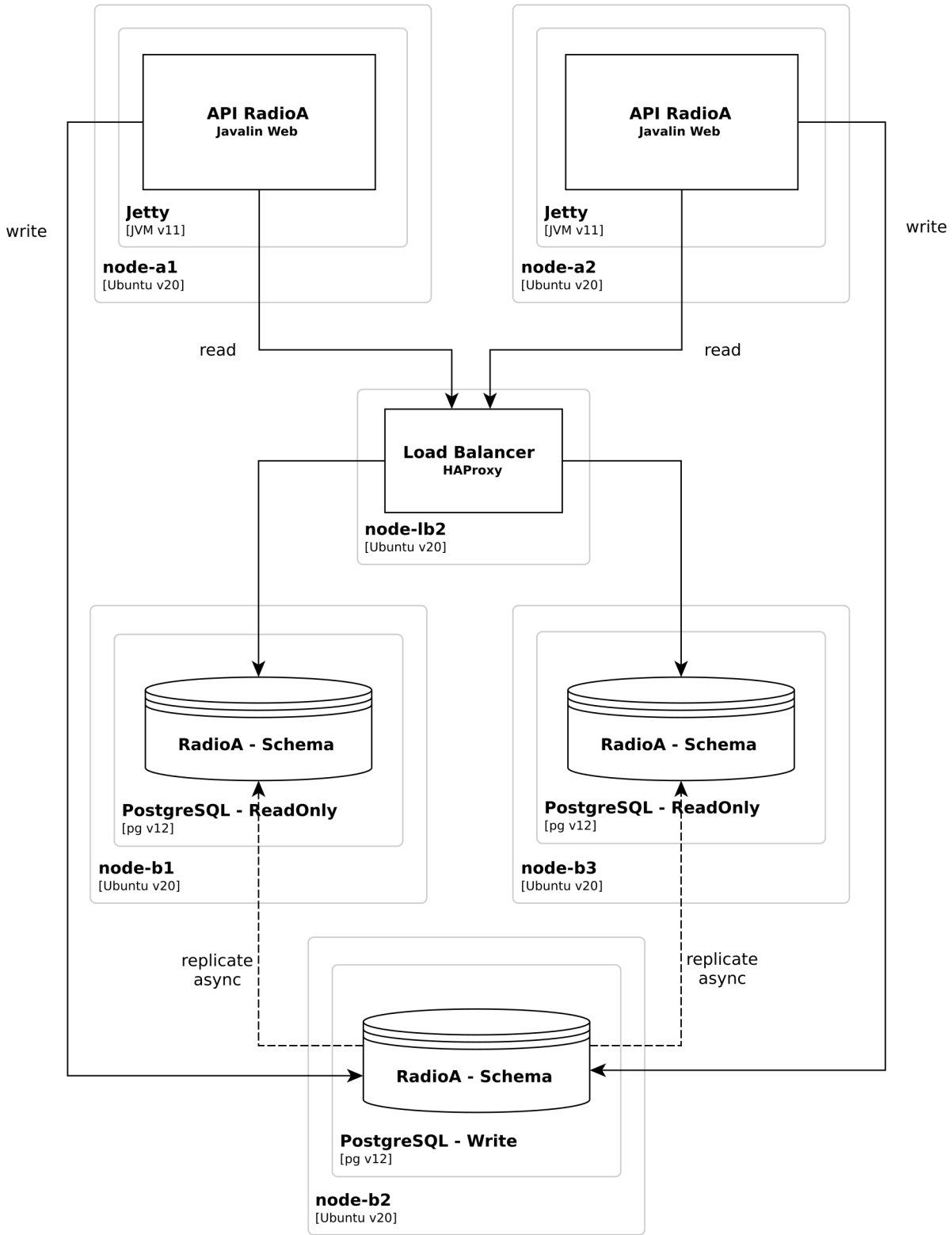

Figure 6.4: Scaling Reads - Deployment Diagram

As can be seen in figure 6.4, there are two application servers that send all the write operations to the *node-b2*, which is the master database. The master database replicates to read-only replicas (*node-b1* and *node-b3*). This database configuration is called master/slave.

If the replication from master to slaves is synchronous we would have a distributed configuration of type **EC**, and if it is asynchronous (like the configuration in figure 6.4) it would be **EL**. On the other hand, for the application to send all the read operations to the slaves databases (*node-b1* and *node-b3*), it's necessary to have some load balancing mechanism (*node-lb2*) as shown in figure 6.4.

Supposing that our master database is relational, using other replication mechanisms like Change Data Capture* we could scale the read operations also replicating to NoSQL databases (in memory or disk). This requires some extra effort (as replication mechanisms offered by relational databases works only for databases with identical models) but it might be necessary if we require some features only available in NoSQL databases.

Note that with this configuration read operations can scale without limit, adding new read replicas as needed. We release the master database from some load caused by read operations and we are able to distribute that load into multiple read replicas.

If we detect that our bottleneck is in the master database (because we have a high number of concurrent write operations), we have to look at a different option. There is a replication configuration called master/master, where all nodes receive read and write operations and all replicate towards all. But this configuration does not work to scale writes, since they all receive the same amount of requests, there isn't load distribution. Although they do work to improve availability. The only way to scale write operations is by using Sharding.

Sharding: Scaling Writes

The only way to scale write operations is to do a **horizontal partitioning** of the data. In this way some sets of records are in a database server and other sets of records are in another database server. This is called **Sharding**.

Figure 6.5 shows one of the existing ways to implement Sharding which is called *range partitioning*. There you can see three databases storing information about people according to the value of the attribute *id*. Each of these databases is called a **shard**. The application needs to store the mapping between the range and the database that contains the people in that range, so that it can identify which shard to access with each operation to read or modify information from people.

*https://en.wikipedia.org/wiki/Change_data_capture

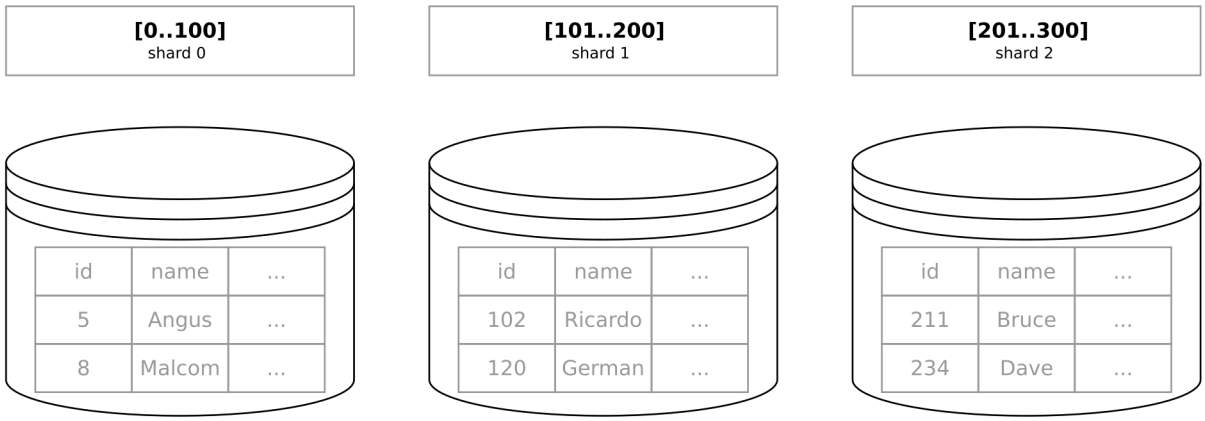

Figure 6.5: Sharding

Another way to implement Sharding is called *hash partitioning*. By applying a *hash* function to a specific attribute, it determines which shard it belongs to. This form does not require storing any mapping, since the calculation is performed dynamically with each request/operation.

It is important to understand that every read or write query must contain the attribute used to identify which shard to send the request to. This of course limits a lot what types of queries can be made. However, as data is distributed across several shards, the load is distributed too.

Scaling a relational database using Sharding is an *extremely* complex task, as all the complexity of partitioning the data and distributing requests goes into the application. NoSQL databases come with what is called **Auto - Sharding**. That is, they are in charge of partitioning, distributing and replicating the data so that the development team does not have to worry about this.

Distributed relational databases[69] [70], with Auto-Sharding, high availability and SQL, have started to emerge. In all of them, of course, the CAP and PACELC theorem applies, so it is important to understand them in order to know what to expect in each case.

Notes

68 D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," Computer, 2012.

69 Yugabyte.

70 MySQL Cluster.